Sydr: Cutting Edge Dynamic Symbolic Execution

Alexey Vishnyakov*, Andrey Fedotov*, Daniil Kuts*, Alexander Novikov*, Darya Parygina*†,
Eli Kobrin*†, Vlada Logunova*‡, Pavel Belecky* and Shamil Kurmangaleev*

*Ivannikov Institute for System Programming of the RAS
†Lomonosov Moscow State University
‡Moscow Institute of Physics and Technology
Moscow, Russia
{vishnya, fedotoff, kutz, a.novikov, pa_darochek, kobrineli, vlada, belecky, kursh}@ispras.ru

Abstract—The security development lifecycle (SDL) is becoming an industry standard. Dynamic symbolic execution (DSE) has enormous amount of applications in computer security (fuzzing, vulnerability discovery, reverse-engineering, etc.). We propose several performance and accuracy improvements for dynamic symbolic execution. Skipping non-symbolic instructions allows to build a path predicate 1.2–3.5 times faster. Symbolic engine simplifies formulas during symbolic execution. Path predicate slicing eliminates irrelevant conjuncts from solver queries. We handle each jump table (switch statement) as multiple branches and describe the method for symbolic execution of multi-threaded programs. The proposed solutions were implemented in Sydr tool. Sydr performs inversion of branches in path predicate. Sydr combines DynamoRIO dynamic binary instrumentation tool with Triton symbolic engine. We evaluated Sydr features on 64-bit Linux executables.

Index Terms—symbolic execution, concolic execution, dynamic analysis, binary analysis, dynamic binary instrumentation, computer security, security development lifecycle, DSE, SMT, DBI, SDL

I. Introduction

We can search errors in programs in various ways: at the compile time, manually, applying static analysis tools to source [1, 2] and binary [3, 4] code, dynamic analysis tools, formal verification tools, etc. Security researchers and developers widely use fuzzing [5, 6], dynamic symbolic execution [7, 8], and systems combining both of them [9–11] to detect errors. The security development lifecycle (SDL) is becoming an industry standard [12–14]. Developers are required to apply various analysis tools to improve the quality of their product. These tools have two purposes: (1) generate new inputs that expand the code coverage; (2) find errors. Of course, while solving (1) problem one can detect a certain number of errors, but it is more efficient to separate these tasks. Based on the experience of developing dynamic symbolic execution tools [15–17], we developed a new dynamic symbolic execution tool (Sydr) that addresses the problem of expanding coverage. We are going to extend Sydr to solve problem of finding errors in future.

Dynamic symbolic execution [18–20] explore variation of initial input data on some fixed execution path. Initially each byte of input data is modeled by a free symbolic variable. Each instruction is modeled with an SMT [21] formula over constants and symbolic variables according to its operational semantics. Symbolic engine maintains a symbolic state that is a mapping from memory bytes and registers to SMT formulas. All symbolic register/memory modifications update the symbolic state with new formulas. Branch conditions on the explored path are represented by SMT formulas and form the path predicate. Thus, the path predicate contains the constraints that describe the explored path. The solution to conjunction of these path constraints is an input data that follows the same execution path. In order to invert some branch, we negate its path constraint.

We symbolically execute a program to invert branches. Thus, we are able to discover new paths that regular fuzzing can hardly reach. The main goal of this work is improving dynamic symbolic execution accuracy and performance. Accuracy is essential, because we need generated inputs to actually invert target branches and discover new paths. Increasing performance helps to gain new inputs faster.

This paper makes the following contributions:

- We surveyed existing symbolic execution improvement approaches and implemented the most promising ones. We evaluated each method to measure its impact on symbolic execution in general. Sections II–VI describe the main features: skipping non-symbolic instructions II, AST simplification III, path predicate slicing IV, indirect jumps resolving V, and handling multi-threaded programs VI. The experimental results presented in Section VIII.

- We present Sydr, a dynamic symbolic execution tool, which implements each of these techniques and can be used as a path explorer for fuzzing tools. We describe the tool design in Section VII.

II. Skipping Non-Symbolic Instructions

Symbolic execution approximately slows down a target application execution between 1000 and 1000000 times. We skip symbolic execution of non-symbolic instructions to build the path predicate faster and reduce memory usage. To determine whether an instruction is symbolic we retrieve all its explicit and implicit (e.g. `pop rax` implicitly reads from the stack and modifies the stack pointer) operands with the help of

DynamoRIO [22, 23] disassembler. Then we symbolically execute the instruction iff any of its read/write registers (including ones used for computing memory address, e.g. `rbx` in `mov rax, [rbx]`), memory, or flags are symbolic. Skipping non-symbolic instructions allows us to build path predicate 1.2–3.5 times faster (Table VI). Moreover, we consume less memory because we create less SMT [21] statements.

III. AST SIMPLIFICATION

Symbolic engines tend to use an intermediate AST representation that is later translated to SMT. Symbolic engine may simplify these ASTs before passing them to solver [8, 24]. We implemented several AST simplifications in Triton [25, 26] symbolic engine. These simplifications improve accuracy of symbolic execution, help solver, reduce memory used by ASTs, and improve readability of printed SMT statements. We list some of them below:

- $A \mathbin{\&} A \to A$, $A \mathbin{|} A \to A$.
- $A \oplus A \to 0$, $0 * A \to 0$, $A - A \to 0$, $0 \mathbin{\&} A \to 0$, $0 \ll A \to 0$, $0 \gg A \to 0$. Triton marks AST as symbolic iff it contains symbolic variables. AST simplifications can remove symbolic marks after kill operations. For instance, symbolic `rax` after `xor rax, rax` will be marked as non-symbolic.
- `((_ extract high low) ((_ extract hi lo) A)) → ((_ extract high+lo low+lo) A)`
- `((_ extract 11 9) (concat (_ bv1 8) (_ bv2 8) (_ bv3 8) (_ bv4 8))) → ((_ extract 3 1) (_ bv3 8))`. Triton represents each symbolic parent register with an AST. Modification of lower register byte results in AST that concatenates extracted register high part with a new byte value. In particular, if we place a constant in lower byte of symbolic register and later extract it, we will get a symbolic AST. The proposed simplification provides a non-symbolic AST. So, `jz` branch in instruction sequence `mov rax, symbolic_variable ; mov al, 0x00 ; test al, al ; jz 0xdeadbeef` won't be symbolic.
- `(concat ((_ extract 31 24) A) ((_ extract 23 16) A) ((_ extract 15 8) A) ((_ extract 7 0) A)) → ((_ extract 31 0) A)`. For instance, this simplification is useful when we store register in memory and later retrieve it. Triton extracts each register byte and stores it to symbolic memory. During the register load Triton concatenates all its extracted parts.
- `((_ extract 31 0) ((_ zero_extend 32) (_ bv1 32))) → (_ bv1 32)`. Instructions that operate on 32-bit general-purpose registers in x86-64 zero extend the result to parent register. If we move one 32-bit register to another and ask for its AST, we are going to get lower 32 bits extraction from zero extended register. The simplification just returns the original moved register.

Algorithm 1 Path predicate slicing algorithm.

Input: $cond$ – predicate for target branch inversion, Π – path predicate (path constraints prior to the target branch).

$vars \leftarrow used\_variables(cond)$ ▷ slicing variables
$change \leftarrow vars$
while $change \neq \varnothing$ **do**
    $change \leftarrow vars$
    for all $c \in \Pi$ **do**   ▷ iterate over path constraints
        if $vars \cap used\_variables(c) \neq \varnothing$ **then**
            $vars \leftarrow vars \cup used\_variables(c)$
    $change \leftarrow vars \setminus change$
$\Pi_S \leftarrow cond$ ▷ predicate for branch inversion
for all $c \in \Pi$ **do**   ▷ iterate over path constraints
    if $vars \cap used\_variables(c) \neq \varnothing$ **then**
        $\Pi_S \leftarrow \Pi_S \wedge c$
return Π_S

IV. PATH PREDICATE SLICING

We use path predicate slicing (a.k.a. constraint independence optimization [27, 28]) to eliminate irrelevant conjuncts from solver queries. For clarity, we define a path predicate Π as a sequence of constraints. Each constraint corresponds to a taken branch condition on the execution trace. Algorithm 1 returns a predicate for inversion of the target branch. Π contains constraints for all taken branches that were executed before the target branch. Function $used\_variables(c)$ returns a bitset of symbolic variable numbers used in constraint c. We initialize slicing variables $vars$ with variables used in predicate for target branch inversion $cond$ (a negation of taken target branch constraint). Then we iterate over and over constraints in Π updating $vars$ with variables that transitively depend on slicing variables. Finally, we conjunct only those branch constraints that have variables from $vars$. Thus, we leave only constraints that are relevant to inverting the target branch.

With slicing applied solver returns a model only for some subset of symbolic variables. We retrieve values for missing symbolic variables from initial input data. The resulting solution is correct due to the fact that initial input data is already a solution for a path predicate.

Slicing allows to perform a more powerful symbolic execution. It has the following advantages:

1) Solver consumes less memory and time to resolve a query. We get model only for a part of input data that is responsible for branch inversion.
2) Undertaint [29] can cause some symbolic variables to be underconstrained. Thus, generated input may not reproduce the desired path. Slicing remove possibly underconstrained symbolic variables from the solver query. These variables values are taken from the initial input.

Consider the following code:

```
1  char* syms = "SLICING_FIX_IT!\n";
```

```
2
3  int main(int argc, char **argv) {
4    FILE *ptr = fopen(argv[1], "rb");
5    unsigned *b = malloc(6 * sizeof(int));
6    fread(b, 6, sizeof(int), ptr);
7    int len = strlen(syms);
8    if (b[0] < len)
9      if (syms[b[0] % len] == '!')
10       if (b[2] > '@')
11         if (b[5] + b[4] < 'B')
12           if (b[3] + b[5] > '@')
13             if (b[1] + b[3] > '@')
14               if (b[4] < '9')
15                 if (b[1] > '@')
16                   printf("OK\n");
17                 else
18                   printf("FAIL\n");
19 }
```

Initial input data leads to printing `FAIL` in line 18. We illustrate how slicing algorithm inverts the branch in line 15. Slicing appends constraints for the following branches to the resulting predicate:

- Line 15 (slicing variable `b[1]`).
- Line 13 (slicing variables `b[1]`, `b[3]`).
- Line 12 (slicing variables `b[1]`, `b[3]`, `b[5]`).
- Line 11 (slicing variables `b[1]`, `b[3]`, `b[4]`, `b[5]`).
- Line 14 (slicing variables `b[1]`, `b[3]`, `b[4]`, `b[5]`).

Line 9 contains a symbolic address `b[0] % len` that will be concretized by the symbolic engine. So, the branch in line 9 is underconstrained (not symbolic). Appending branch 8 to predicate without 9 may result in invalid `b[0]` model. Slicing skips branch 8. Thus, we successfully generate an input causing our example program to print `OK`. If we run the example on input generated without slicing, condition in line 9 won't hold.

V. INDIRECT CONTROL TRANSFERS RESOLVING

Handling indirect control flow transfers is crucial for the complete and accurate program analysis. For the branch inversion problem, we are only interested in table control flow transitions. In such cases, the target jump address is taken from an array of pointers located in the program memory. The offset for the corresponding array element is computed from the branch condition. Jump tables [30] are generated by compilers from the long switch and if-else statements for the optimization purpose. Furthermore, compiler also produces jump tables for function pointer arrays. Besides the direct code pointers, jump tables may contain values (address offsets) that take part in computing the target jump address:

```
lea rdx, [rax * 4]
lea rax, [rip + 0x155]
mov eax, [rdx + rax]
movsxd rdx, eax
lea rax, [rip + 0x148]
add rax, rdx
jmp rax
```

The assembly above is a typical indirect jump. In the first two lines the table index and the base address are calculated. In the next two lines a value loaded from the jump table and stored in `rdx` register. Then this value is added to the computed target base address and then a jump is made to the resulting address.

To dereference a symbolic pointer Triton [25, 26] gets its value from the concrete state. We propose indirect control transfer resolving to partially handle symbolic pointers.

We consider each jump on address calculated from memory cell as a potential indirect control transfer instead of determining the table control flow dependencies by code patterns. We perform backward slicing [31] within a current basic block to detect such transitions. It's a trivial case when jump/call instruction has memory reference operand. If target operand is a register, we start tracking this register data flow dependencies up to the beginning of the basic block. Thus, we locate an instruction reading value from memory that forms the jump target register.

Firstly, we check whether jump table exists at the previously detected instruction memory access address. We support two kinds of tables which contain addresses or offset values. Address tables should contain values that are valid executable addresses. We use heuristic for the offset tables. An offset should be a negative double word value. At least one of adjacent memory cells should contain value of the same type, otherwise it will not be interpreted as a valid jump table. We assume that table is continuously located in memory. We parse memory in both directions from the current access while the conditions for corresponding table type are met. Parsing stops upon reaching the configurable maximum table size limit. Besides, the valid jump table should contain at least three entries.

Sometimes several different jump tables are placed in memory continuously. In such cases the exact table bounds cannot be determined, and the maximum size limitation prevents memory parsing overhead. As a result, part of the current jump table is missed and part of adjacent jump table is parsed. Missed part of jump table can be parsed during re-execution of analyzed program with another memory access address. And those jump table entries, which belong to another indirect jump will produce incompatible path constraints during branch inversion.

After successful jump table parsing, we generate path constraints for the indirect jump. A condition for each branch is an equality of the symbolic pointer expression and the corresponding jump table entry address. Since the indirect jumps are usually compiled from switch statements, some jump table entries can point to the same jump target. We create only one path constraint for each unique jump target to prevent generating several different inputs for the one jump direction. The conditions for duplicated targets are merged with disjunction.

If jump table contains offset values, we should calculate jump table targets separately. Usually target addresses for such cases are computed as some base address plus offset value from the table. Knowledge of concrete target address for the current execution and offset value for the current branch helps to determine this base address. Then destination addresses for every jump table branch can be easily deduced by adding offset

to this computed base.

VI. SYMBOLIC EXECUTION OF MULTI-THREADED PROGRAMS

The regular analysis of multi-threaded programs corrupts symbolic model and makes the following symbolic execution incorrect. To be able to analyze such programs, we need to keep track of thread switching and maintain separate symbolic states for each thread.

All threads within one process have all memory shared but their own register values, so this should be considered in symbolic model. All threads can also have a shared path predicates storage for symbolic branches, because they are built on different symbolic registers values and won't affect each other.

The process of saving and restoring registers on the control flow transition between threads is called a context switch. In order to handle multi-threaded programs we implemented a context switching operation on symbolic model. We maintain a thread contexts storage that contains symbolic registers for each thread. On each thread switching we save all symbolic registers and replace them with symbolic registers for the current thread.

The proposed technique application can be considered on the following program:

```
1  int d[20], mins[4], P[4] = {0, 1, 2, 3};
2
3  void *min(void *thread_number) {
4    int i = *((int *) thread_number);
5    int cnt = sizeof(d) / sizeof(*d) / 4;
6    mins[i] = d[i * cnt];
7    for (int j = i*cnt+1; j < (i+1)*cnt; ++j)
8      if (mins[i] > d[j]) mins[i] = d[j];
9  }
10
11 int main(int argc, char **argv) {
12   int fd = open(argv[1], O_RDONLY);
13   read(fd, d, sizeof(d));
14   pthread_t t[4];
15   for (int i = 0; i < 4; ++i)
16     pthread_create(t+i,0,min,(void*)&P[i]);
17   for (int i = 0; i < 4; ++i)
18     pthread_join(t[i], 0);
19   int m = mins[0];
20   for (int i = 1; i < 4; ++i)
21     if (m > mins[i]) m = mins[i];
22   if (m > 100) printf("min>100");
23 }
```

This example implements a minimum search in an input array. In lines 15–16 four threads are created. Each thread searches a minimum for a part of an input array. Afterwards, the main thread computes a global minimum. We symbolically execute this program on array with minimum less than 100. In lines 6–8 local minimums are stored to the shared symbolic memory. In lines 19–21 the main thread computes a global minimum. We are inverting the branch in the line 22. Thus, the program prints min>100. Furthermore, the example illustrates that dynamic symbolic execution is a path-sensitive analysis. The generated input will not only have numbers greater than 100,

but also it will satisfy all constraints from branches in lines 8 and 21. These constraints actually define a partial order on an input array. If we don't switch symbolic registers, we get additional unsound path constraints and lose some essential constraints on input array. It is due to the fact that symbolic registers get overwritten by ones from the other thread. Thus, some array elements may be less than 100 and the generated input won't invert the target branch.

The limitation of this approach is that we don't influence the thread order during program execution. A generated input may not follow an expected path. To solve this problem it is necessary to implement a thread scheduler to arrange threads order [32]. We may address this problem in future.

VII. IMPLEMENTATION

We implemented the improvements described above in Sydr (Symbolic DynamoRIO) tool. Sydr is a dynamic symbolic execution (DSE) tool based on dynamic binary instrumentation (DBI). Sydr performs symbolic execution along one path (defined by input data) and generates new inputs that invert branches discovered on that path.

There are two approaches for implementing DSE: (1) collect execution trace and perform symbolic execution using that trace [17, 33]; (2) perform symbolic execution while program is executed [34]. The (1) method has an overhead for storing execution trace on hard drive and processing the trace to generate SMT formulas. The technique (2) doesn't have overhead for storing traces on disk, but it is also challenging. DBI allows to insert analyzing code before every executed instruction. This instrumentation code may drive symbolic execution. Such analysis is limited to 4GB RAM when it is applied to 32-bit executables. Moreover, problems may occur when your instrumentation code is complex and it uses some external libraries. For instance, DynamoRIO client crashes when it is linked with pthread library [35]. Furthermore, DynamoRIO heap is quite slow [36]. These kind of problems motivated us to separate concrete and symbolic execution into two processes. This separation allows to reduce concrete executor code. Symbolic executor is not limited to 4 GB RAM and can be linked with any libraries needed for analysis.

Fig. 1 presents Sydr architecture. Sydr implements (2) method for symbolic execution. Concrete and symbolic execution are separated into two processes communicating via shared memory. *Concrete Executor* places *events* in shared memory that are later processed by *Symbolic Executor*.

Concrete Executor (Conex) has two components: *Input Detector* and *DynamoRIO*. *Input Detector* recognizes system calls and library functions that handle input data specified by user. When such system call is detected *Conex* sends an event to *Symbolic Executor (Symex)*. This event (ReadSymbolicInput) holds information used by *Symex* to create new symbolic variables. WriteSymbolicInput event allows to track symbolic variables when data are stored to disk. DynamoRIO implements dynamic binary instrumentation. *Input Detector* also requires DBI to hook system and
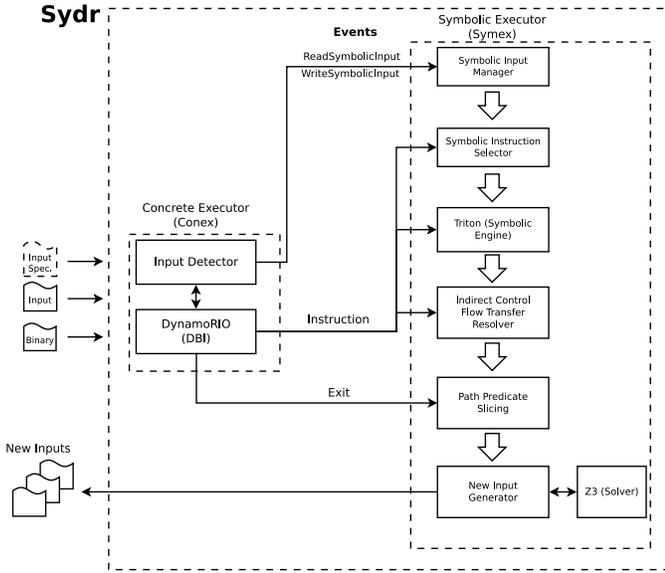
Fig. 1. Sydr architecture.

library calls. DBI collects information about executed instruction: address, opcode, explicit and implicit operands with their concrete values. This information is sent as `Instruction` event to *Symex* for symbolic execution.

Symbolic Executor (Symex) handles events from *Conex* to perform symbolic execution. *Symbolic Input Manager* is responsible for creating symbolic variables. It updates symbolic registers, memory, and files states when `Read/WriteSymbolicInput` events occur. Manager also contains concrete values of input data corresponding to symbolic variables. These values are needed while producing new inputs. When *Conex* detects first read from symbolic input, it starts passing `Instruction` events to *Symex*. These events firstly go through *Symbolic Instruction Selector*. Selector chooses just instructions having at least one symbolic operand (explicit or implicit). These selected instructions are executed symbolically by Triton. *Indirect Control Flow Transfer Resolver* detects indirect control transfer instructions, determines possible control flow target addresses, and constructs path predicate constraints for them. `Exit` event passed to *Symex* signals that concrete execution is stopped. At this point path constraint building is finished and *Path Predicate Slicing* component starts to perform algorithm described in **Path Predicate Slicing** section. *New Input Generator* component inverts branch conditions in path predicate (including indirect control flow jumps/calls) to produce new inputs. To invert each branch a corresponding SMT solver (we use Z3 [37, 38]) query is formed. In each query we only use those symbolic variables that affect a target branch to be inverted, i.e. the other parts of input data stay unchanged. The set of inputs from *New Input Generator* is provided to user.

Sydr supports parallel inversion of branches. We build a complete path predicate first and then solve SMT queries in parallel threads. Moreover, we terminate each solver query

by a specified timeout. We could have been inverting some branches during the path predicate construction, but further research is needed.

VIII. Evaluation

We evaluated Sydr on a set of single-threaded 64-bit Linux executables [39]. We leave multi-threaded programs evaluation for future research. For evaluation we used the server with the following specification: processor AMD EPYC 7702 (128 cores), 256G RAM. We also checked the correctness of generated inputs. If new input has the same execution trace as original except the last branch, that should be in inverted direction, this input is correct. We developed a tool based on DynamoRIO to verify the inputs correctness. In tables below column named **Correct** represents the number of correct inputs. The column named **SAT** shows the amount of satisfiable solver **Queries**. Each query is an attempt to invert branch (change control flow direction). The column named **Branches** is the number of symbolic branches in path predicate. It should be noted that the total number of branches can be less than the number of queries, because each jump table is considered as one branch and produces multiple queries. Sydr inverts branches from first to last in path predicate. Each test is executed up to 2 hours. We limit path predicate construction time to 20 minutes.

TABLE I
RESULTS WITH ALL PROPOSED TECHNIQUES

Application	Correct	SAT	Queries	Branches	Time
bzip2recover	2101	2101	5131	5131	51m3s
cjpeg	50	50	197	8010	120m
faad	426	430	652	458145	120m
foo2lava	27	31	6127	910725	120m
hdp	809	1037	3828	67476	120m
jasper	6766	6798	18207	837669	120m
libxml2	545	1069	17532	53699	120m
minigzip	3896	7569	8977	8977	29m42s
muraster	3227	3228	4726	7102	120m
pk2bm	182	183	3673	3673	21m39s
pnmhistmap_pgm	17088	17089	25446	967187	120m
pnmhistmap_ppm	106	107	8247	8121	28m52s
readelf	639	739	6141	64196	120m
yices-smt2	2114	2699	9647	19543	120m
yodl	180	313	5201	4831	34m59s

We present results with all proposed techniques in Table I. Then we disable some method in order to determine its influence.

Table II contains results without path predicate slicing. Slicing significantly increases accuracy of generated inputs. For some programs (*jasper*, *minigzip*, *hpd*, *pnmhistmap* with .pgm file, *yices-smt2*, *readelf*) the amount of correct branches increased in several times with path predicate slicing. Still, for *bzip2recover* and *cjpeg* result stays the same.

Table III presents results with all proposed techniques and randomly chosen branches. The amount of correct inputs for tests fitting in 2 hour limit is the same. For other tests the

TABLE II
RESULTS WITHOUT PATH PREDICATE SLICING

Application	Correct	SAT	Queries	Branches	Time
bzip2recover	2101	2101	5131	5131	52m42s
cjpeg	50	50	198	8010	120m
faad	386	389	585	470588	120m
foo2lava	27	31	6252	910725	120m
hdp	116	464	2427	67475	120m
jasper	1	1987	5639	837669	120m
libxml2	130	1043	13520	53700	120m
minigzip	425	3961	4183	8977	120m
muraster	3234	3235	4739	7102	120m
pk2bm	181	183	3672	3673	21m48s
pnmhistmap_pgm	3158	3159	4681	967187	120m
pnmhistmap_ppm	106	107	8247	8121	40m15s
readelf	135	218	2046	64196	120m
yices-smt2	13	521	2135	19543	120m
yodl	26	313	5201	4831	43m24s

TABLE III
RESULTS WITH ALL PROPOSED TECHNIQUES (BRANCHES ARE
RANDOMLY CHOSEN)

Application	Correct	SAT	Queries	Branches	Time
bzip2recover	2101	2101	5131	5131	51m31s
cjpeg	6	6	425	8010	120m
faad	18	71	1234	470588	120m
foo2lava	1	1	19	910725	120m
hdp	24	48	427	67476	120m
jasper	34	34	127	837669	120m
libxml2	26	62	2918	53699	120m
minigzip	3896	7569	8977	8977	18m26s
muraster	102	125	246	7102	120m
pk2bm	182	183	3672	3673	21m47s
pnmhistmap_pgm	537	537	819	967187	120m
pnmhistmap_ppm	106	107	8247	8121	27m
readelf	61	70	833	64196	120m
yices-smt2	1193	1543	5811	19543	120m
yodl	180	313	5201	4831	39m3s

number of correct inputs decreased because the complexity of
solver queries for randomly chosen branches increased.

TABLE IV
PARALLEL SOLVING (CORRECT INPUTS)

Application	Number of Threads			
	1	2	4	8
cjpeg	50	54	113	113
faad	426	507	582	2803
foo2lava	27	27	32	32
hdp	809	1052	1441	1813
jasper	6766	9746	11965	12269
libxml2	545	545	545	545
muraster	3227	3450	3896	3968
pnmhistmap_pgm	17088	19861	24360	24115
readelf	639	1244	1652	2018
yices-smt2	2114	3002	4147	4157

We evaluated how parallel solving influences on input
generation. We ran benchmark with all proposed techniques
using 1, 2, 4, and 8 solving threads. Table IV shows how

TABLE V
PARALLEL SOLVING (TIME)

Application	Number of Threads			
	1	2	4	8
bzip2recover	51m3s	25m57s	13m35s	10m12
cjpeg	—	—	63m12s	24m8s
minigzip	29m42s	17m18s	9m13s	6m49s
pk2bm	21m39s	11m21s	5m47s	3m1s
pnmhistmap_ppm	28m52s	14m20s	7m34s	4m14s
yodl	34m59s	16m54s	9m14s	5m23s

parallel solving increases the amount of correct generated
inputs. This table displays tests that do not fit in 2 hour limit
and don't invert all branches in path predicate. For *libxml2* test
the number of correct inputs stayed the same, but the number
of queries increased from 17532 to 38092. Degradation of
results for *pnmhistmap_pgm* (8 threads worse than 4 threads)
could be explained by exhaustion of all CPU cores. We ran
several tests in parallel. Tool for testing the correctness of
inputs ran in parallel too. Table V represents how time needed
for analysis decreased with parallel solving. There are only
tests that fit in 2 hour time limit and invert all branches in
path predicate. We can see that test named *cjpeg* using 4 or
more threads fits in 2 hour time limit.

TABLE VI
PATH PREDICATE CONSTRUCTION TIME

Application	Input Size	Branch Count	App Time	Path Predicate Time		
				Base	Skip	X
bzip2recover	147b	5131	0.0018s	9s	5s	1.8
cjpeg	12K	8010	0.0017s	39s	16s	2.4
faad	33K	470588	0.0082s	46m35s	18m7s	2.6
foo2lava	34K	910725	0.0045s	22m32s	18m42s	1.2
hdp	530K	67478	0.0021s	1m6s	41s	1.6
jasper	198K	837669	0.0037s	—	14m11s	—
libxml2	453b	53699	0.0024s	1m5s	34s	1.9
minigzip	19K	8977	0.0023s	2m44s	58s	2.8
muraster	887b	7102	0.0024s	7s	3s	2.3
pk2bm	1.7K	3673	0.0018s	4s	2s	2.0
pnmhistmap_pgm	198K	967187	0.0038s	14m37s	7m55s	1.8
pnmhistmap_ppm	12K	8121	0.0021s	29s	11s	2.6
readelf	8.3K	64196	0.0019s	1m19s	36s	2.2
yices-smt2	2K	19543	0.0029s	26s	14s	1.9
yodl	280b	4831	0.0017s	21s	6s	3.5

The Table VI contains evaluation of path predicate con-
struction time. The column **App Time** represents the running
time of the program without instrumentation. The column **Base**
shows running time without skipping non-symbolic instruc-
tions. Running time with skipping non-symbolic instructions
is presented in the column **Skip**. Skipping non-symbolic
instructions makes path predicate building 1.2–3.5 times faster.
Path predicate building for *jasper* didn't complete for 24 hours.
We should investigate the reasons for that later.

Results of the tool application without indirect control flow
transfers are shown in Table VII. Only a few tested programs
have symbolic indirect jumps: *faad*, *muraster*, *readelf*, *yices*,

Application	Correct	SAT	Queries	Branches	Time
bzip2recover	2101	2101	5131	5131	51m8s
cjpeg	50	50	197	7986	120m
faad	427	431	653	422272	120m
foo2lava	27	31	6119	910725	120m
hdp	815	1050	3851	67383	120m
jasper	6572	6604	17710	837670	120m
libxml2	545	1085	16232	53548	120m
minigzip	3896	7569	8977	8977	31m34s
muraster	2652	3861	4998	6018	120m
pk2bm	181	183	3673	3673	21m30s
pnmhistmap_pgm	17062	17063	25410	967187	120m
pnmhistmap_ppm	106	107	8058	8058	27m
readelf	629	727	5815	64093	120m
yices-smt2	2056	2596	9183	19386	120m
yodl	159	275	4795	4795	33m47s

yodl. Other programs can be used to evaluate how pure jump table detection algorithm affects the analysis performance.

The programs with indirect jumps have more detected symbolic branches. New inputs that lead to previously undiscovered paths were generated for these programs. The largest number of indirect jumps was found on *muraster* – near a 1000 symbolic branches. These queries were quite difficult for solver to process, so there are less processed queries in the same time. On the contrary, the number of incorrectly generated inputs (that do not actually invert the target branch) has significantly decreased. Without processing indirect jumps, the program path predicate is missing some of the branches that depend on the input. Inversion of a branch after such missed jumps will not consider its condition, which in turn will lead to the generation of the wrong input. For instance, we cannot correctly invert branch in a switch case. Thus, resolving indirect control transfers allows to increase an analysis accuracy.

The number of discovered branches and processed queries for programs without symbolic indirect jumps did not change significantly. Therefore the implemented jump table detection mechanism does not reduce the tool performance.

IX. FUTURE WORK

We plan to continue research in improving dynamic symbolic execution. There are several interesting areas to research:

- Modeling *function semantics* in symbolic execution could increase accuracy and possibly speed up DSE (`tolower`/`toupper` are interesting because they constrain a symbol case).
- *Symbolic memory model* [9] could provide new symbolic states interesting for futher analysis.
- Using *Z3-solver tactics* could possibly decrease time spent in solver.
- Developing *light-weight security predicates* to find some types of dangerous vulnerabilities.

We have already partially developed security predicates, which find critical errors, such as null pointer dereference and out of bounds access vulnerabilities. In future we plan to research integer overflow, wraparound, and some other dangerous types of critical defects.

Besides, during evaluation we independently found bugs in some programs [40, 41], including commercial NTFS support module for UEFI. Furthermore, we plan to improve bug detection in our analysis: iteratively launching application on various inputs, testing generated inputs for hangs.

X. CONCLUSION

We have presented Sydr, a tool for dynamic symbolic execution that embodies the best techniques to analyze real world programs. We designed it in a way to provide an independence from restrictions imposed by instrumentation platform and target programs. Our evaluation results showed that all considered methods are crucial for accuracy and performance. The symbolic engine ASTs simplification and skipping execution of non-symbolic instructions enhance analysis efficiency. Path predicate slicing, indirect control transfer resolving, and maintenance of thread-based symbolic states allows us to significantly increase the analysis soundness and expand the boundaries of our tool applicability.

REFERENCES

[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010. DOI: 10.1145/1646353.1646374.

[2] V. P. Ivannikov, A. A. Belevantsev, A. E. Borodin, V. N. Ignatiev, D. M. Zhurikhin, and A. I. Avetisyan, "Static analyzer Svace for finding defects in a source program code," *Programming and Computer Software*, vol. 40, no. 5, pp. 265–275, 2014. DOI: 10.1134/S0361768814050041.

[3] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "CodeSurfer/x86—a platform for analyzing x86 executables," in *Compiler Construction*, Springer Berlin Heidelberg, 2005, pp. 250–254. DOI: 10.1007/978-3-540-31985-6_19.

[4] H. Aslanyan, M. Arutunian, G. Keropyan, S. Kurmangaleev, and V. Vardanyan, "BinSide : Static analysis framework for defects detection in binary code," in *2020 Ivannikov Memorial Workshop (IVMEM)*, IEEE, 2020, pp. 9–14. DOI: 10.1109/IVMEM51402.2020.00007.

[5] S. Sargsyan, J. Hakobyan, M. Mehrabyan, M. Mishechkin, V. Akozin, and S. Kurmangaleev, "ISP-Fuzzer: Extendable fuzzing framework," in *2019 Ivannikov Memorial Workshop (IVMEM)*, IEEE, 2019, pp. 68–71. DOI: 10.1109/IVMEM.2019.00017.

[6] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020. [Online]. Available: https://www.usenix.org/system/files/woot20-paper-fioraldi.pdf.

[7] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 1–49, 2012. DOI: 10.1145/2110356.2110358.

[8] 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 *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157. DOI: 10.1109/SP.2016.17.

[9] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on binary code," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12, IEEE Computer Society, 2012, pp. 380–394. DOI: 10.1109/SP.2012.31.

[10] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, vol. 16, 2016, pp. 1–16.

[11] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium*, 2018, pp. 745–761. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-yun.pdf.

[12] M. Howard and S. Lipner, *The security development lifecycle*. Microsoft Press Redmond, 2006, vol. 8. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms995349.aspx.

[13] *ISO/IEC 15408-3:2008: Information technology – security techniques – evaluation criteria for it security – part 3: Security assurance components*, ISO Geneva, Switzerland, 2008. [Online]. Available: https://www.iso.org/standard/46413.html.

[14] *GOST R 56939-2016: Information protection. secure software development. general requirements*, National Standard of Russian Federation, 2016. [Online]. Available: http://protect.gost.ru/document.aspx?control=7&id=203548.

[15] I. K. Isaev and D. V. Sidorov, "The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs," *Programming and Computer Software*, vol. 36, no. 4, pp. 225–236, 2010. DOI: 10.1134/S0361768810040055.

[16] A. Gerasimov, S. Vartanov, M. Ermakov, L. Kruglov, D. Kutz, A. Novikov, and S. Asryan, "Anxiety: A dynamic symbolic execution framework," in *2017 Ivannikov ISPRAS Open Conference (ISPRAS)*, IEEE, 2017, pp. 16–21. DOI: 10.1109/ISPRAS.2017.00010.

[17] V. A. Padaryan, V. V. Kaushan, and A. N. Fedotov, "Automated exploit generation for stack buffer overflow vulnerabilities," *Programming and Computer Software*, vol. 41, no. 6, pp. 373–380, 2015. DOI: 10.1134/S0361768815060055.

[18] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976. DOI: 10.1145/360248.360252.

[19] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 317–331. DOI: 10.1109/SP.2010.26.

[20] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys*, vol. 51, no. 3, 2018. DOI: 10.1145/3182657.

[21] C. Barrett, P. Fontaine, and C. Tinelli, *The SMT-LIB Standard: Version 2.6*, 2017. [Online]. Available: www.SMT-LIB.org.

[22] D. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004. [Online]. Available: https://www.burningcutlery.com/derek/docs/phd.pdf.

[23] ——, *DynamoRIO: Dynamic instrumentation tool platform*. [Online]. Available: https://github.com/DynamoRIO/dynamorio.

[24] *Claripy: An abstraction layer for constraint solvers*. [Online]. Available: https://github.com/angr/claripy.

[25] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications*, ser. SSTIC, 2015, pp. 31–54. [Online]. Available: https://triton.quarkslab.com/files/sstic2015_slide_en_saudel_salwan.pdf.

[26] J. Salwan, *Triton: Dynamic binary analysis framework*. [Online]. Available: https://github.com/JonathanSalwan/Triton.

[27] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06, ACM, 2006, pp. 322–335. DOI: 10.1145/1180405.1180445.

[28] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, vol. 8, 2008, pp. 209–224. [Online]. Available: https://static.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

[29] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic taint analysis with targeted control-flow propagation," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS '11, 2011.

[30] C. Cifuentes and M. Van Emmerik, "Recovery of jump table case statements from binary code," *Science of Computer Programming*, vol. 40, no. 2, pp. 171–188, 2001. DOI: 10.1016/S0167-6423(01)00014-4.

[31] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984. DOI: 10.1109/TSE.1984.5010248.

[32] S. Guo, M. Kusano, and C. Wang, "Conc-ISE: Incremental symbolic execution of concurrent software," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016, ACM, 2016, pp. 531–542. DOI: 10.1145/2970276.2970332.

[33] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS*, vol. 8, 2008, pp. 151–166. [Online]. Available: https : / / www . microsoft.com/en‑us/research/publication/automated‑whitebox-fuzz-testing/.

[34] D. A. Molnar and D. Wagner, "Catchconv: Symbolic execution and run-time type inference for integer conversion errors," UC Berkeley EECS, Tech. Rep. UCB/EECS-2007-23, 2007. [Online]. Available: https://digitalassets.lib.berkeley.edu/techreports/ucb/text/EECS-2007-23.pdf.

[35] D. Bruening, *Issue for DynamoRIO libpthread support*. [Online]. Available: https://github.com/DynamoRIO/dynamorio/issues/2848.

[36] ——, *Issue for DynamoRIO heap slowdowns*. [Online]. Available: https://github.com/DynamoRIO/dynamorio/issues/2115.

[37] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, 2008, pp. 337–340. DOI: 10.1007/978‑3‑540‑78800‑3_24.

[38] L. De Moura and N. Bjørner, *The Z3 theorem prover*. [Online]. Available: https://github.com/Z3Prover/z3.

[39] *Sydr benchmark*. [Online]. Available: https://github.com/ispras/sydr-benchmark.

[40] *Goblin bug*. [Online]. Available: https://github.com/m4b/goblin/issues/108.

[41] *Faad2 bug*. [Online]. Available: https://github.com/knik0/faad2/pull/65.