Synthesising Programming Languages

Yuxi Ling

National University of Singapore

Synopsis

- **Programming Language Synthesis** (PLS): a set of techniques for automatically generating a *high-level programming language* from a restricted set of *low-level instructions* with custom semantics.
- **Application**: Automated generation of executable exploits via *Data-Oriented Programming* (DOP) [3].

Data-Oriented Programming in a Nutshell

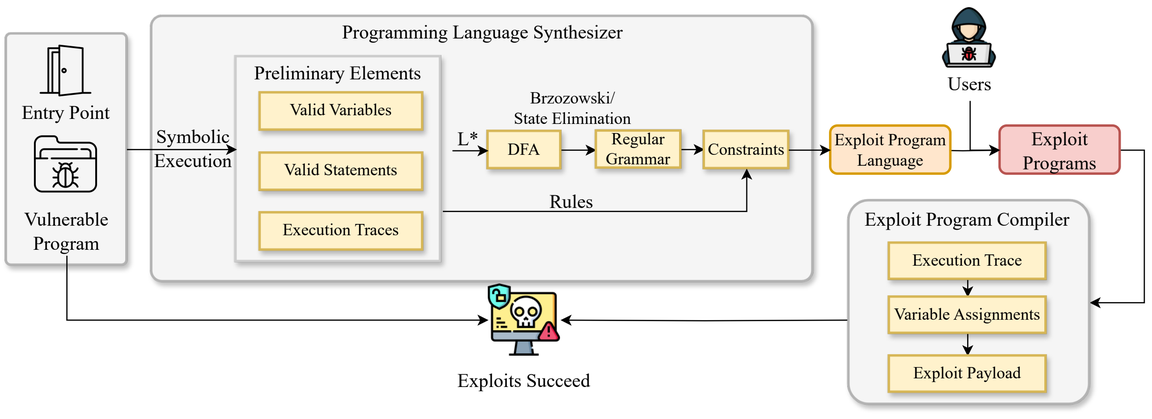
Consider this C program with a classical vulnerability:

```
1  int m, n, p, q; int *a, *b, *c;
2  char buf[1024];
3  ...
4  while (m--) {
5      gets(buf); // buffer overflow: can override variables at line 1
6      if (n == 0)
7          printf("%d", *a);
8      else if (n > 10)
9          *b = p;
10     else if (n > 5)
11         *c += q;
12 }
```

An attacker can manipulate the variables `m`, `n`, `p`, `q`, *etc*, thereby executing the program's arithmetic operations, assignment operations, and arbitrary memory reads. DOP is resilient against general defenses targeting control flow hijacking, such as including CFI, DEP, and ASLR.

Our motivation: Study DOP from the Synthesis perspective.

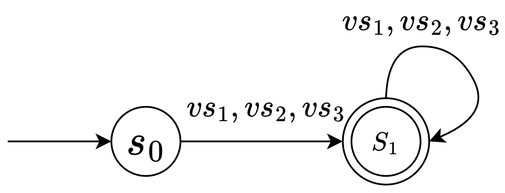
Our Approach: Doppler



Step 1: From Vulnerabilities to Automata

Doppler adapts Angluin's L* Algorithm [1] to learn a deterministic finite automaton (DFA) from a set of vulnerable code's elements:

- *Valid variables*: variables controlled by attackers (*e.g.*, `m`, `n` above)
- *Valid statements*: instructions using valid variables
- *Attack traces*: reachable traces containing valid statements



An automaton for the vulnerable program above.

Step 2: From Automata to Grammars

Doppler applies the state elimination method and Brzozowski algorithm [2] to convert the DFA to the *attack grammar*:

Value	Val	integers
Valid variables	Var	a, b, c, p, q
Valid statements	VS	$::= init : Var = Val$
		$\mid \; vs_1 : init; *b = p$
		$\mid \; vs_2 : init; *c = *c + q$
		$\mid \; vs_3 : init; print(\text{"\%d"}, *a)$
Attack	$attack ::=$	$(vs_1 + vs_2 + vs_3)*$

Step 3: Writing High-Level Programs

Doppler compiles a program written in a high-level attack grammar to the *payload* for valid variables in the vulnerable program.

Two examples of attack programs in our attack grammar (left) and the respective pseudo-code (right):

Example 1: Arbitrary Memory Read

vs_3 (init a to addr of secret variable)	int* i = & secret; print(*i);

Example 2: Fibonacci Sequence

$vs_1; vs_1$ (init b to addr of i, j)
(init p to 1)
(
vs_3 (init a to addr of j, get value of j)
vs_1 (init b to addr of t, p to value of j)
vs_3 (get value of i)
vs_2 (init c to addr of j, q to value of i)
vs_3 (get value of t)
vs_1 (init b to addr of i, p to value of t)
 	
)$^*n-2$ (repeat in $n-2$ times)
vs_3 (init a to addr of j)
	fibonacci(n){
	i = 1; j = 1;
	
	for(k=2; k$\leq$ n; $k++$){
	
	t=j;
	
	j=j+i;
	
	i=t;
	}
	
	print(j);
	}

Compiling: Doppler maps an attack program back to an attack trace in the original program, solves the path constraint by symbolic execution, and constructs a concrete exploit payload.

Future Work

- **Formally proving realisability**: for *any* program written in an attack grammar, there must be a valid exploit payload to produce a semantically equivalent attack trace in the vulnerable program.
- **Formally proving PLS completeness**: we discover the space of *all* exploits that can be executed via DOP on a vulnerable program.
- **Increasing expressivity**: going beyond regular languages.

References:

[1] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[2] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[3] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *USENIX Security Symposium*, pages 177–192, 2015.