USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Sound and Efficient Generation of Data-Oriented Exploits via Programming Language Synthesis

Yuxi Ling, *National Univeristy of Singapore;* Gokul Rajiv, *National University of Singapore;* Kiran Gopinathan, *University of Illinois Urbana-Champaign;* Ilya Sergey, *National University of Singapore*

https://www.usenix.org/conference/usenixsecurity25/presentation/ling

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

Sound and Efficient Generation of Data-Oriented Exploits via Programming Language Synthesis

Yuxi Ling* Gokul Rajiv* Kiran Gopinathan† Ilya Sergey*

*National University of Singapore
†University of Illinois Urbana-Champaign

Abstract

Data-oriented programming (DOP) is a methodology for embedding malicious programs into fixed executable vulnerable binaries. DOP is effective for implementing code reuse attacks that exploit memory corruptions without violating many defence techniques, such as non-execute, address space layer randomisation, control flow and code point integrity. Existing approaches for automated exploit generation for DOP follow the *program synthesis* approach: given a description of an attack phrased as a program, they perform extensive constraint-based search to identify the required payload for the corrupted memory. The program synthesis-inspired approaches come with three major shortcomings regarding (a) *efficiency*: attack generation often takes prohibitively large amount of time, (b) *soundness*: they provide no formal guarantees whatsoever that a particular user-described attack is *feasible* in a particular vulnerable program with suitable payloads, and (c) *capability visibility*: they do not make clear to users what attack capabilities are admitted by the vulnerable program.

In this work, we propose a novel approach to synthesise code reuse attacks via DOP by casting this task as an instance of the previously unexplored programming *language* synthesis idea. Given a vulnerable program and an exploit (*e.g.*, buffer overflow), our approach derives a *grammar* of a programming language for describing the available attacks. Our approach addresses the issue (a) by shifting the cost of synthesising individual attacks to synthesising the entire attack language: once the grammar is generated, the compilation of each attack takes negligible time. The issues (b) and (c) are addressed by establishing correctness of our grammar synthesis algorithm: *any* attack expressible in terms of a generated grammar is realisable. We implement our approach in a tool called DOPPLER—an end-to-end compiler for DOP-based attacks. We evaluate DOPPLER against available state-of-the art techniques on a set of 17 case studies, including three recent CVEs, demonstrating its improved effectiveness (it generates more attacks) and efficiency (it does so much faster).

1 Introduction

Data-Oriented Programming (DOP) [29, 30] is an approach to generate code reuse attacks by exploiting memory corruptions such as buffer overflows. A security expert can construct a DOP-based attack by calculating the *data payload* for the corrupted memory segment in a way that would predictably impact the control-flow of the vulnerable program, thus, making it execute a desired functionality (*e.g.*, writing a particular value to a desired memory location or printing out a message), thereby realising the attack. For vulnerable programs with relatively complex control-flow patterns, DOP has been shown to be Turing-complete in expressivity. That is, it allows for embedding arbitrarily complex attacks via suitable payloads, while circumventing common control-flow hijacking defences [30,32], such as Data Execution Prevention (DEP) [20], Control Flow Integrity (CFI) [1, 9, 13], Address Space Layout Randomisation (ASLR) [55, 56], Stack Canaries [19], and dynamically loaded libraries (Libsafe) [60, 61].

DOP attacks were extensively studied recently [4, 16, 32, 45, 53] from the perspective of automatic exploit generation (AEG) [5] by applying ideas from automated program synthesis [25]. The state-of-the-art approaches for DOP-based attack synthesis either supply a fixed relatively high-level domain-specific language, in which users can define their attack as a program [32, 45] or the desired memory state they would like to achieve [33, 53], as well as a "compiler" that generates the attack by calculating the required data payload for the vulnerable program. Given a vulnerable program, a high-level description of the attack, and additional tool-specific inputs, such as the entry point and the collection of gadgets, all those tools perform a search for the payload by employing a combination of concolic execution and constraint solving.

Because of their search-based nature (hence the quotes around the "compiler" above), all these tools share the following practical and conceptual shortcomings. First, the attack compilation by means of payload generation takes large amount of time, ranging from seconds to days [53], depending on the size of the attack description and the nature of the vul-

nerable program. Due to the opportunistic nature of the search tools, no estimates can be given about the synthesis time. The efficiency problem is exacerbated by the fact that every attack synthesis tasks with such tools starts "from scratch", and there is no way to reuse the results from the previous searches to amortise the time it takes to compile multiple different attacks on top the same exploited program.

Second, none of the existing approaches provide any *soundness* guarantees *wrt. realisability* of the attacks. In other words, there is no way to predict, without running the tool (and possibly waiting for days), whether, for a particular attack, the required payload can even be found for a vulnerable program in question. The lack of methodology to reliably tell whether a certain attack *can* be synthesised with a given approach hinders utility of such tools for security experts.

Our approach and key ideas. To address the shortcomings of state-of-the-art approaches to DOP-based attack generation, we adopt a different mindset. Instead of considering attack generation as a *program synthesis* task, we view it as an instance of the novel idea we call *Programming Language Synthesis* (PLS): given a low-level runtime L and the desired syntax of a high-level language H with (perhaps, informally) specified semantics, produce a correct and efficient compiler from (a subset of) H to L. We say that the implementation of PLS is *sound* if the compiler it generates can compile *all* programs written in the synthesised subset of H (described as a grammar) to semantically equivalent representations in L.

Coming back to AEG challenge for DOP, we can consider the vulnerable program with the set of *all* possible payloads one can deploy in its corrupted state as the language L. Our goal is to discover the human-readable language H and derive a compiler that can compile from it to L by finding, for each program in H, the required concrete payloads in L. By shifting the synthesis task from individual attacks (*i.e.*, programs in H) to *generating the language and the compiler for it*, our methodology addresses the fundamental shortcomings of the existing approaches outlined above. Specifically, it greatly improves the efficiency of the attack generation by providing the compiler that does *not* perform, for each attack, the search in the space of possible symbolic execution traces, as this task has been done *once and for all* during the compiler synthesis phase. Furthermore, correctness of our language synthesis algorithms guarantees *soundness* of the compilation in the following sense: *any* attack expressible in the derived grammar can be compiled to the equivalent payload, resulting in the desired outcome of the vulnerable program's execution.[1]

Importantly, by learning the attack grammar, users gain knowledge of the attack capability of a vulnerability. In real-world scenarios, such a sound attack grammar is helpful for security engineers to understand the *severity* of a reported vulnerability; it is also useful for white-hats who want to build and report multiple proof-of-concept attacks in terms of different attack goals, such as arbitrary memory write, remote code execution, *etc*. Finally, the ability to compose attacks via the synthesised language allows for more complex exploits, *e.g.*, using an arbitrary memory read to steal the name of a secret file, subsequently accessing it through a reverse shell.

Contributions. We make the following contributions:

- Our main conceptual contribution is the novel idea of Programming Language Synthesis (PLS), an approach to amortise the cost of synthesising individual programs by synthesising a *language* for encoding computations that *can* be encoded and executed given the specifics of the back-end runtime, and a compiler for this language.
- Our first technical contribution is the instantiation of the PLS idea for DOP attacks by identifying a family of techniques based on symbolic execution and algorithms for automata learning to synthesise, from a given vulnerable program and an entry point in it, a language grammar for supported attacks as well as a compiler for payload generation to realise any program written in this language.
- Our second technical contribution is DOPPLER, a framework for automatic exploit generation of DOP attack by PLS via symbolic execution and automata learning. Given a vulnerable program, DOPPLER generates a domain-specific language for describing the space of realisable attacks.
- Our final contribution is a extensive evaluation of DOPPLER against BOPC [32], the only available state-of-the-art tool for generating arbitrary DOP exploits, on 17 vulnerable programs (6 characteristic examples and 11 real-world programs) under 5 different attack goals. In particular, our case studies include 3 recent CVEs reported in 2024. The results demonstrate tangible benefits of PLS for AEG: a greatly reduced time for exploit generation and the increased effectiveness compared to a program-synthesis based approach.

2 Overview

We present an overview of DOPPLER by means of an illustrative example: generating a program-specific language for performing DOP attacks against a vulnerable target in the presence of strong code-reuse defences, such as CFI and DEP. The generated language, expressed as a grammar, is regular, and is synthesised by DOPPLER through a combination of symbolic execution and automata learning. Attackers can quickly construct attacks for the vulnerable target by writing programs in this language, and DOPPLER will compile these programs to payloads that execute the desired computation.

In the rest of this narrative, we will assume the target program is open-sourced and contains a known memory corruption vulnerability and a set of manually identified vulnerable variables. The execution environment is protected by general

[1] The reader might also think of a different notion of soundness in application to attack synthesis, namely, that any high-level attack program and its compiled counterpart *always behave the same*. Stating this notion of soundness requires precise semantics for both our high- and low-level languages and is largely outside of the scope of this work. Our extensive evaluation allows us to claim that our approach is sound in this sense in practice.

```
1  int m, n;   // control variables
2  int p, q, *a, *b, *c;  // data variables
3  char buf[1024];
4  int secret;
5  ...
6  while(m--){
7      gets(buf); // stack buffer overflow
8      if(n == 0)
9          printf("%d",*a); // syscall
10     else if(n>10)
11         *b = p;  // assignment
12     else if(n>5)
13         *c += q;  // arithmetic operation
14 }
```

Fig. 1: A program with a buffer overflow

code-reuse defences, including CFI, DEP, and ASLR, against control flow hijacking attacks, such as ROP, JOP, and ret2lib.

2.1 DOP and Attack Grammar in a Nutshell

Fig. 1 presents our running example: a simulated web server with a memory corruption that is vulnerable to DOP attacks.

This program implements a fairly simple server loop. It repeatedly reads data from the user into a buffer `buf` (line 7) and then performs one of a number of operations, including printing (line 9), memory writes (line 11) and limited arithmetic operations (line 13). Since the call at line 7 does not implement bounds checks, large inputs will overflow the buffer and overwrite the stack variables at lines 1 and 2, allowing for a data-oriented programming attack.

Given a memory corruption error, like the buffer overflow at line 7, data-oriented programming (DOP) manipulates the program memory to perform bespoke computations. For example, corrupting the value of variables p, q, a, b, c will allow an attacker to exploit the unaltered commands of the program to perform arbitrary memory reads and writes (lines 9 and 11), or restricted computations such as addition (line 13). Furthermore, by corrupting the value of variables m, n, attackers can modify the control flow of the program and can chain these operations in arbitrary sequences. As DOP attacks of this kind do not require modifying control data such as function return addresses, they can easily bypass control-flow protection defences such as CFI, making them a potent attack vector.

The main challenge with constructing DOP attacks in practice is that they are progam-specific and must be tailored for each program and vulnerability. For example, in our running example, a careful inspection of the source code is required to identify both (a) which computations can be encoded by modifying vulnerable variables, and (b) which variables can be modified to chain these computations together. Traditionally this analysis has been done manually, and while there has been some work on automating DOP attacks, no prior work has tried to describe *the landscape of possible attacks* for a target program. In particular, following our analysis from before, one can see that the space of computations allowed by our example is captured by the following grammar:

$$
\begin{aligned}
VS &::= *b = p \\
&\mid *c = *c + q \\
&\mid print(\text{"\%d"}, *a) \\
P &::= VS^*
\end{aligned}
$$

In this work, we propose an approach to constructing DOP attacks by incorporating the program-specific nature of the task and generating bespoke *attack grammars* that describe the space of allowed computations that can then be consistently compiled into viable DOP attacks. A program-specific attack grammar is a high-level language that expresses potential DOP attacks in the target program. It indicates the variables and statements available to the attacker and the rules dictating how they can be composed. Unlike existing works that can only check whether specific attacks are possible on a case-by-case basis [32, 53], grammars synthesised by our tool DOPPLER summarise many possible attack behaviours for a given program/vulnerability, thus, giving the user an understanding of the space of possible attacks.

The remainder of the section will walk through the process through which DOPPLER, when applied to our running example (Fig. 1), generates an attack grammar and then compiles programs in this grammar into an executable attack.

2.2 Preliminary Analysis

Given the code and the location of the vulnerability, the first step in our analysis is to extract the set of variables and statements that will be accessible to the user in the final grammar.

The final variables exposed in the attack grammar will be a subset of those that can be corrupted through the memory vulnerability, in this case m, n, p, q, a, b, c. As the set of corruptible variables may rely on domain-specific details of the memory vulnerability to ascertain, our analysis assumes that an initial set of such variables is supplied by the user. However, not all of these variables are suitable to be presented to the user; for example, in our running example, the variables `m,n` are used to control the number of iterations, and so allowing the user to mutate or use them would interfere with the control flow of the program. As such, DOPPLER first classifies these variables into two types: *control variables*, which influence branch conditions and *data variables*, which are any remaining variables. In constructing an attack program, control variables are used to chain a sequence of target instructions, while data variables are used to construct specific operations. In our example, m and n would be *control variables* while p, q, a, b, c are *data variables*. These variables are identified through a standard backwards data-flow analysis which will be detailed in Sec. 3.

Having identified the appropriate data variables, DOPPLER then traverses all the instructions in the target program and identifies a set of *valid statements*—-operations in the source code that involve data variables and no control variables. The role of these valid statements is largely similar to con-

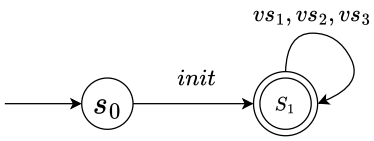

Fig. 2: DFA from Automaton Learning

cept of *data-oriented gadgets* as introduced in prior work on DOP [30]: each valid statement is a potential atomic operation in attack grammar. For our running example, there are three valid statements: $\{printf(\text{"}\%d\text{"}, *a), *b = p, *c \mathrel{+}= q\}$.

2.3 Extracting an Automaton

With a set of valid statements in hand, the next step in our analysis is determining how they can be sequenced within the control-flow graph (CFG) of the program by manipulating the control variables that have been identified earlier.

In particular, the sequencing of valid statements within the program will depend on three key factors: (1) their positions in the CFG, (2) the program variables they depend on, and finally (3) the satisfiability of the path constraint along an execution path. To convert these constraints into simple compositional rules that can be presented in our attack grammars, we first construct a deterministic finite automaton (DFA) to represent the sequences of valid statements allowed by the program.

DFAs are a mathematical formalism of finite state machines and serve as concise representations of regular languages. A DFA is typically represented using a 5-tuple $(Q, \Sigma, \sigma, q_0, F)$, where Q represents the set of states of the machine, Σ the alphabet over which it runs, $\sigma : Q \times \Sigma \to Q$ the transition relation, q_0 the initial state, and $F \subseteq Q$ the set of accepting states. To use this formalism to encode traces of the program, DOPPLER regards the states of the target program as the finite set of states Q, valid statements as symbols Σ, the accept states F being the states reached by any valid execution traces (sequences of valid statements). The initial state q_0 represents the program entry point, and transition function σ, to be learned, is the transition incurred by executing a valid statement.

To construct a DFA that will represent the valid traces of our program, DOPPLER adapts Angluin's L* Algorithm [3] to *learn* it from observations of concrete traces. In particular, in Angluin's algorithm, a DFA can be learned through a number of membership and equivalence queries to an oracle.[2] In our implementation, DOPPLER uses a symbolic execution engine, more specifically, KLEE [14], as such an oracle. When prompted, the symbolic execution engine can then provide DOPPLER with a number execution traces that could be feasible. If this trace contains the previously identified valid statements, then DOPPLER can learn that these sequences of statements are feasible and should be matched by the DFA. So far, this only produces positive examples, so, in order to constrain the DFA to reduce the number of incorrectly accepted traces, DOPPLER also synthesises negative counterexamples, mutating existing traces, using KLEE to validate them, and passing them to Angluin's algorithm.

[2] The reader might notice that using L* can result in unsoundness of PLS, since the resulting automaton can also accept traces that do not correspond to program executions. For our experiments (*cf.* Sec. 4), we use passive learning [40], which guarantees soundness. DOPPLER allows for using L* as well, which results in more expressive yet possibly unsound grammars.

Value	*Val*		integers
Valid Variables	*Var*		a, b, c, p, q
Initialisers	*init*	::=	$Var = Val$
Valid Statements	*vs1*	::=	$*b = p$
	vs2	::=	$*c = *c + q$
	vs3	::=	$print(\text{"}\%d\text{"}, *a)$
Attack	*attack*	::=	$init; (vs1 + vs2 + vs3)^*$

Fig. 3: Attack grammar for the example from Fig. 1

The DFA for Fig. 1 is shown in Fig. 2, where *init* is the initial statement to setup variables, and $\{vs_1, vs_2, vs_3\}$ correspond to the three valid statements outlined above.

2.4 From Automata to Grammars

Once a DFA has been constructed, the last step is to map it to an attack grammar. To do this, DOPPLER applies the state elimination method [12] to convert the DFA to a regular expression, representing the compositional rules of valid statements in the attack grammar. It then uses a number of basic rewrite rules to normalise and simplify the resultant expression and map it to a grammar. For our running example, applying this process produces the grammar shown in Fig. 3.

In this grammar, there are five available variables that can be assigned to arbitrary variables by *init* statement, three valid statements with an *init* statement appended in the front. Intuitively, before the execution of valid statements, all the valid variables can be initialised by attackers through buffer overflow. DOPPLER finds that $(vs_1 + vs_2 + vs_3)$ can be repeated an arbitrary number of times, thus, introducing the regular expression remains $(vs_1 + vs_2 + vs_3)*$ to the resulting grammar.

2.5 Compiling High-Level Programs

Given a computation written as a program a synthesised attack grammar, DOPPLER also provides a pipeline to compile it to concrete exploit payloads that will use DOP execute the intended computation. This is fairly straightforward: DOPPLER maps the user program to an execution trace in the DFA and then to a path in the original CFG, and then constructs the path condition for this input, which can be solved by an SMT solver to learn the required values for the exploit payload.

To illustrate the utility of our grammars for attack construction, in the rest of this section we will explore two examples of performing non-trivial computations in our running example from Fig. 1: the first one executes an arbitrary memory read, while the other implements an attack program with complex logic, calculating an arbitrary Fibonacci number.

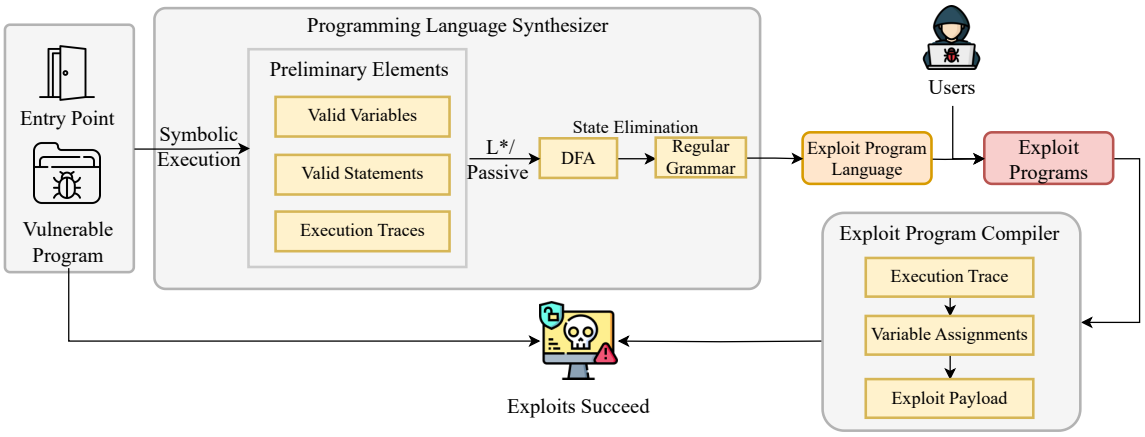

Fig. 4: An overview of DOPPLER: a programming language synthesis framework and an attack compiler.

Example 1: Executing arbitrary memory read. Tab. 1 shows the attack program and semantically equivalent pseudocode that reads a variable *secret* from memory. By changing the pointer value of valid variable a in the *init* statement, an arbitrary memory read attack is constructed.

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

Tab. 1: A program written in attack grammar reading variable *secret* from memory and an equivalent pseudocode.

Example 2: Calculating a Fibonacci number. Tab. 2 shows the attack program and pseudocode that implements a function to calculate the Fibonacci sequence for a given number of n. The program is contains three pointer variables (a, b, c), one assignment operation (vs_1), one addition operation (vs_2), and one print call (vs_3). To fully realise this exploit requires a degree of interactivity since some variables need to be re-assigned at runtime (further discussed in Sec. 5).

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

Tab. 2: A program written in attack grammar calculating Fibonacci sequence and an equivalent pseudocode.

Our approach guarantees that attack programs written using the synthesised grammar are *realisable*. That is, for any program written in attack grammar, there *must* be a valid exploit payload to produce an equivalent execution trace in the target program (see details in Sec. 4).

2.6 Putting It All Together

Fig. 4 illustrates a high-level overview of our framework, DOPPLER. It takes the source code and entry point as inputs, where the entry point specifies the location of the memory corruption and a set of sensitive variables. The programming language synthesiser conducts a static program analysis and symbolic execution to identify a set of valid variables, valid statements, and execution traces, called preliminary elements. Based on this, PLS constructs a DFA through automaton learning. By converting DFA to regular grammar and restricting it with composition rules, DOPPLER synthesises the exploit programming language. Users write attack programs in this language, while our exploit program compiler calculates actual exploit payloads. Finally, it sends the payload to the original vulnerable program, realising the encoded attack.

3 Algorithms and Framework Design

The core problem solved by DOPPLER is an instance of *Programming Language Synthesis*: given a language of low-level commands L, the syntax of a desired high-level language H, and the semantics of both, produce a correct compiler from (a subset of) H to L. Intuitively, we reuse the low-level backend, that is the source code of the vulnerable program, for the execution of attack programs in H by compiling H to L. The key step in PLS is to abstract the logic of a restricted subset of the language L and summarise it in terms of H. In this work, we focus on performing *sound* program language synthesis in the sense that *all* the programs written in the resulting high-level language H should always be possible to be compiled to equivalent programs in L.

Algorithm 1 Identifying Valid Variables

Input: P : target program, S : source variables
Output: C : control variables, D : data variables

```
1:  I ← instructions of P
2:  C, D ← ∅
3:  V ← ∅                                          ▷ valid variables
4:  for i ∈ I do
5:      if depends(S, i) then
6:          V.add(i)
7:          if i is a conditional branch then
8:              C.add(v)
9:              while i' = i.pre() do
10:                 if depends(i', i) ∧ depends(S, i') then
11:                     C.add(i')                  ▷ backwards analysis
12: C = unique(C)
13: D = V \ C
14: return C, D
```

Note that this is an important difference between the languages produced by DOPPLER and related works, such as SPloit Language (SPL) in BOPC [32], MinDOP in DOP [29], and SLANG in Steroids [45]. DOPPLER gives a unique attack grammar for each target program/vulnerability, and programs written in that grammar are always realisable. In contrast, other tools cannot provide this guarantee; instead, they check the existence of a payload for a specific attack goal written in the given language, one at a time.

3.1 Preliminaries

Valid Variables. Given a target program P, valid variables *VVar* are those whose value directly or indirectly derives from one or more of the variables that are controlled by attackers at the starting memory vulnerability. Intuitively, a variable is *valid* if its value changes under different assignments of sensitive variables set in the entry point. We classify valid variables into two categories based on their positions and related variables. We say that a variable is a *control variable* if it is used in a branch condition or depends on any valid variables used in a branch condition. A valid variable is said to be a *data variable* if it is not a *control variable*.

Algorithm 1 shows the procedure of identifying control and data variables using static program analysis. We compile the source code of the target program to LLVM's intermediate representation (IR) language and perform the labelling of valid variable at the LLVM IR level. We assume the user provides a set of source variables that attackers can manipulate directly through the memory vulnerability. For example, in the program shown in Fig.1, source variables are $\{m,n,p,q,a,b,c\}$. The key idea to construct the set of valid variables is to iterate over instructions in the control flow of P and annotate all the variables whose value depends on source variables. The function *depends()* at lines 6 and 10 implements a standard dependence analysis [24]. Whenever a valid variable is found in a conditional branch, it is marked as a control variable and a backwards search is invoked to retroactively mark any valid variables that it depends on as control. Finally, all remaining valid variables that are not labelled as control variables are marked as data variables. In the final attack grammar, only data variables are exposed to users, while control variables are used for chaining statements.

Algorithm 2 Identifying Valid Statement

Input: P : program, D : data vars, C : control vars
Output: M : valid statements

```
1: M ← ∅
2: I ← instructions of P                          ▷ in DFS order
3: for i ∈ I do
4:     V_i ← variable symbols in i
5:     if V_i ∩ D ≠ ∅ ∧ V_i ∩ C = ∅ then
6:         M.add(i)
   return M
```

Valid Statements. Similarly to the notion of data-oriented gadgets in prior works [30], we introduce a notion of *valid statements*—instructions from P that may be used to construct DOP attacks. Given a set of data variables D, the set of *valid statements* are instructions from the target program that involve at least one data variable and no control variables. In particular, we exclude instructions using control variables from the set of valid statements to avoid potential conflicts between user programs and the control flow.

Algorithm 2 shows the relatively straightforward procedure for identifying the set of valid statements using static program analysis. We traverse the control flow graph and iterate over instructions in P, labelling instructions with at least one data variable and no control variables as valid statements.

Execution Traces. We perform symbolic execution during the analysis to extract execution traces from the target program. Each execution trace consists of a sequence of valid statements indicating a potential attack trace. Execution traces provide information about how valid statements can be sequenced together under certain exploit payloads.

3.2 Automata Learning

To extract underlying sequential composition rules of valid statements from execution traces and compile them into a human-readable grammar, a finite automaton serves as an effective intermediate representation both due to its operational effectiveness and the existence of a minimal form. This brings the problem to a well-known *regular* language learning challenge equivalent to automata learning [27]. Specifically, our goal is to synthesise a regular grammar where the set of valid statements can be regarded as alphabets, and execution traces are a finite set of strings of our alphabet.

Algorithm 3 Adapted L^* Algorithm

```
1:  Input: Σ : alphabet; S : seed strings; N : a threshold
2:  Output: M : DFA
3:  S ← {λ}                                          ▷ prefixes
4:  E ← {λ}                                          ▷ suffixes
5:  T ← ∅                                            ▷ observation table
6:  while True do
7:      T ← checkMembership(S, E, T)
8:      while True do
9:          if isClosed(T) ∧ isConsistent(T) then
10:             break
11:         T ← updateObservationTable(T)
12:     M′ ← makeDFA(S, E, T)
13:     ce ← checkEquivalence(M′, N)                 ▷ pass N
14:     if ce is None then                           ▷ ce: counterexample
15:         return M′
16:     addPrefixes(ce, S)
```

Why a regular language? Our automata learning step aims to build a deterministic finite state automaton from preliminary elements through automata learning.

In practice, it is possible to formulate programs whose exploit grammars cannot be expressed by regular expressions (such as $vs_1{}^n; vs_2{}^n$) nor even context-free grammars (such as $vs_1{}^n; vs_2{}^m; vs_3{}^n; vs_4{}^m$). However, symbolic execution provides us with a finite number of execution traces, which serve as positive samples for automata learning. Unless the symbolic execution provides a complete set of execution traces (which would be finite), it is impossible to tell if a trace outside the set of positive samples is reachable or not. It is thus not possible to *soundly* (in the sense of realisability) infer a grammar that is irregular from the execution traces alone.

Therefore, in this work, we explore generation of regular languages leaving synthesis of other language classes, such as context-free ones, for in future. To do so, we use two learning algorithms: passive [40] and Angluin-style L* [3]. The two algorithms implement different trade-offs in terms of soundness and expressivity (*cf.* Sec. 3.2.3), and can be used in DOPPLER interchangeably depending on the user's goals.

3.2.1 Passive Learning

Given an alphabet set Σ and a finite set of strings $S = \{x_1; x_2; ...; x_n\}$, $x_i \in \Sigma^*$, passive learning [40] is one of the most straightforward methods to construct a DFA M that accepts exactly string set S. We do this by first constructing an non-deterministic finite automaton (NFA) that represents the regular expression $x_1 + x_2 + ... + x_n$. Next, we convert the NFA to a DFA by subset construction algorithm [27]. By further minimising the DFA through state partitioning strategy, we get a more compact DFA with fewer states. The average time complexity for DFA minimisation is $O(n \log n)$.

3.2.2 L* Algorithm

Active learning is an effective approach to construct a DFA M compatible with the string set S by *interactively* querying an oracle under an assumption that the oracle "knows" the final result and can provide an infinite number of counter-examples.[3] We apply Angluin's L* algorithm and regard results from symbolic execution as the (incomplete) oracle or the teacher as shown in Algorithm 3. Specifically, we construct membership and equivalence queries as follows:

- **Membership queries**: Whether a string is in the language is equivalent to checking whether the path that it corresponds to, is reachable by some program execution.
- **Equivalence queries**: Whether the hypothesis automaton M' equal to the target M is difficult to answer with the positive samples from symbolic execution alone. We apply a negative counterexample generation strategy to achieve an approximate equivalence checking. Specifically, we build a counterexample set C of predetermined size N, by finding strings of increasing length that are not in S. By ensuring that all strings in S are accepted and those in C are not, we obtain an approximate equivalence between M' and M.

3.2.3 Soundness and Completeness Trade-Off

DOPPLER constructs a DFA that accepts the set of strings arising from the traces obtained through symbolic execution. The passive learning approach produces a DFA that accepts *exactly* this set of strings. Since the set of traces is finite, the DFA is acyclic. The soundness of the regular grammar corresponding to this DFA follows from the soundness of the symbolic execution. However, the string set generated from execution traces is possibly incomplete and is limited by the symbolic execution engine. Thus, the *completeness* of the grammar through passive learning depends on the representativeness of the traces. The L* active learning approach might produce a more complete but unsound DFA that will accept all strings in the set and possibly some other strings. In particular, active learning could produce cyclic DFAs, which cannot be ascertained with a finite symbolic execution.

These approaches present a trade-off between realisability and completeness of the grammar. If we aim to capture more possible attacks through active learning, the soundness of the grammar cannot be guaranteed. Conversely, if we prioritise soundness by passive learning, it will affect completeness (*i.e.*, expressivity). DOPPLER accommodates both options and gives users the ability to choose the learning algorithm. Our experiments (Sec. 4) are conducted using passive learning.

3.3 Grammar Synthesis

The target exploit language is a regular language, defined as a tuple, $L = (\Sigma, N, q_0, P)$, where Σ is the terminal or alphabet set, N is non-terminal symbols, q_0 is the initial state, and P is the

[3] In practice, such an oracle is under-approximated with testing [38].

set of transition rules. It is straightforward to convert a DFA to a left or right linear regular grammar [27]. DOPPLER supports the conversion from regular language to regular expression through state elimination method [12], which is an equivalent but more compact format to present the grammar.

3.4 Payload Compilation

Given a regular language describing the target program/vulnerability capability *wrt.* DOP attacks, users can read and understand the regular grammar and write their attack programs in it. Each attack program corresponds to a specific execution path within the target program. By retrieving the execution path from the attack program, the symbolic execution engine solves the path constraint and returns a set of assignments for the vulnerable variables, which is crucial in constructing the final payload.

3.5 Implementation

DOPPLER is implemented in C++ in a total of about 6,600 lines of code. It integrates KLEE [14] to conduct symbolic execution and is built on top of LLVM 13, targeting vulnerable C programs. DOPPLER takes four inputs: (1) the vulnerable program compiled to LLVM IR; (2) the source code of the vulnerable program with annotations for the corruptable variables (*e.g.*, those affected by a particular buffer overflow) for KLEE to treat them symbolically; (3) a JSON file that lists corruptable variables and their positions in the source; (4) an entry function where function inlining and symbolic execution starts. The entry function is optional for DOPPLER inputs and defaults to `main` if not specified.

The initial outputs are (a) a grammar in the format discussed in Sec. 3.3 and (b) a compiler—a binary file with the suffix *.doppler*. The user is then expected to write an attack program using the synthesised grammar; the accompanying generated compiler will check it for syntax errors and output a sequence of tuples describing how corruptable variables should be set to construct a payload. We leave it to the user to set those variables accordingly by, *e.g.*, providing the required input to the function. We will show several end-to-end examples of using DOPPLER for executing exploits in Sec. 4.1.

4 Evaluation

In this section, we evaluate (1) how effectively and (2) how efficiently DOPPLER can generate exploit grammars using our programming language synthesis approach, and (3) how good are the grammars it is capable of synthesising. As a baseline for the comparison *wrt.* the aspects (2) and (3), we use BOPC by Ispoglou *et al.* [32]. In summary, we seek to answer the following research questions:

- **RQ1**: How effective is DOPPLER? To answer this question, we deploy DOPPLER on 17 programs with memory corruption vulnerability, evaluate whether attack goals are realisable and whether attack grammar correctly describes the DOP attack capability, and compare it with BOPC.[4]
- **RQ2**: How efficient is DOPPLER? To evaluate DOPPLER, we have run it and BOPC on the same benchmarks and compared their execution times.
- **RQ3**: How expressive are the grammars generated by DOPPLER? We analysed the diversity of statement types, grammar size, and the power of the regular expression for statement concatenation provided by the grammar.

Attack Goal	Description
memrd	Read from arbitrary memory
memwr	Write to arbitrary memory
print	Print a message from memory
nloop	Perform a loop in n iterations
summation	Calculate the sum of first n natural numbers
linkedlst	Create a linked list in the memory

Tab. 3: Attack goals and their textual descriptions

Benchmarks. We first design 6 demonstration programs with relatively legible control flow to illuminate DOPPLER features. Then, we select 11 real-world programs, where 8 of them are selected from related works [29, 30, 32, 33, 53] and 3 are recent CVEs reported in 2024. Benchmark programs include user utility (*sudo*), server applications (*proftpd*, *nginx*, etc) and database systems (*redis* and *sqlite*), which are common targets in real-world attack scenarios.

To better adapt LLVM 13 and KLEE, we have the following simplifications to all 11 real-world programs: (1) adjusting configuration files to make them compilable in LLVM 13. (2) semantics preserved rewrite in code and structures that KLEE cannot analyse; (3) pruning some program branches irrelevant to attackable variables. That is, all experiments in Sec. 4 are conducted using versions of the programs with these simplifications applied. Importantly, these simplifications would not compromise the effectiveness of DOPPLER's results, ensuring that payloads describing a sequence of variable assignments from the simplified versions *remain applicable to the original programs*. We will illustrate these with case studies presented towards the end of Sec. 4.1.

Experimental Setup. All our experiments are conducted on a commodity laptop running Ubuntu 22.04, with Intel Core i7 processor, 16GB of RAM, and 512GB of disk space.

[4] Two other closely related tools are Steroids [45] and Limbo [53]. Unfortunately, we could not obtain either of these tools for our experiments, even by emailing their authors: in one case the implementation of the tool has been lost forever due to server cleansing, in another case the tool could not be shared with us by its authors because of licensing issues.

	Program	memrd		memwr		print		nloop		summation		linkedlst	
		DOPPLER	BOPC	DOPPLER	BOPC	DOPPLER	BOPC	DOPPLER	BOPC	DOPPLER	BOPC	DOPPLER	BOPC
1	demo-1	✓	✗$_{0.38}$	✓	✗$_{0.35}$	✓	✗$_{0.36}$	–	–	–	–	–	–
2	demo-2	✓	✗$_{0.36}$	✓	✗$_{0.38}$	✓	✗$_{0.35}$	✓	✗$_{0.36}$	–	–	–	–
3	demo-3	✓	✗$_{0.41}$	✓	✗$_{0.41}$	✓	✗$_{0.40}$	–	–	✓	✗	–	–
4	demo-4	✓	✗$_{0.40}$	✓	✗$_{0.41}$	✓	✗$_{0.40}$	–	–	–	–	✓	✗
5	demo-5	✓	✗$_{0.36}$	✓	✗$_{0.38}$	✓	✗$_{0.37}$	✓	✗$_{0.35}$	✓	✗	–	–
6	min-dop	✓	✗$_{1.63}$	✓	✗$_{1.52}$	✓	✗$_{1.50}$	✓	✗$_{1.51}$	–	–	–	–
7	proftpd	✓	✗$_{2.87}$	✓	✗$_{2.97}$	✗	✗$_{3.14}$	✓	✗$_{2.87}$	–	–	–	–
8	ghttpd	✓	✗$_{3.24}$	✓	✗$_{3.39}$	✗	✗$_{3.31}$	✓	✗$_{3.31}$	–	–	–	–
9	nullhttpd	✓	✗$_{8.84}$	✓	✗$_{8.34}$	✗	✗$_{8.06}$	✓	✗$_{8.12}$	–	–	–	–
10	sudo	✓	✗$_{39.3}$	✓	✗$_{26.6}$	✗	✗$_{24.6}$	✗	✗$_{24.3}$	–	–	–	–
11	httpd	✗	✗$_{3.53}$	✓	✗$_{3.29}$	✗	✗$_{3.38}$	✗	✗$_{3.30}$	–	–	–	–
12	nginx	✓	✓$_{1817}$	✓	∞	✗	✗$_{512}$	✗	✗$_{321}$	–	–	–	–
13	sqlite	✗	∞	✓	∞	✓	✗$_{187}$	✓	✗$_{185}$	✓	✗	✓	✗
14	redis	✓	✗$_{14.1}$	✓	✗$_{14.7}$	✓	✗$_{14.8}$	✓	✗$_{16.1}$	–	–	–	–
15	cherry	✓	✗$_{3.42}$	✓	✗$_{3.47}$	✓	✓$_{3.66}$	✗	✗$_{3.29}$	–	–	–	–
16	pico	✓	✗$_{2.08}$	✓	✗$_{2.11}$	✓	✗$_{2.08}$	✗	✗$_{2.21}$	–	–	–	–
17	hcode	✓	✗$_{5.71}$	✓	✗$_{5.65}$	✓	✗$_{5.64}$	✗	✗$_{5.57}$	–	–	–	–

Tab. 4: Performance of DOPPLER and BOPC on vulnerable programs for a number of attack goals. ✓for DOPPLER means variable assignments that achieve the attack goal were generated given an initial set of source variables and entry point, while ✗ for DOPPLER means variable assignments were not found. ✓for BOPC means a payload (GDB script) was generated for the SPL program while ✗ indicates failure. ∞ means it took longer than 2 hours. BOPC results include subscripts for execution time (or time until it reports failure) in seconds if the attack goal can be expressed in the SPL language. – means the attack doesn't exist.

4.1 RQ1: Effectiveness of DOPPLER

Attack Goals. We first define six attack goals as shown in Tab. 3. The first three, `memrd`, `memwr`, and `print` are common attacks in exploiting memory corruption errors, allowing attackers to access and manipulate arbitrary memory space: `memrd` is reads from an arbitrary memory address, `memwr` writes an arbitrary value to an arbitrary memory address, and `print` prints a value to `stdout` from an arbitrary memory address. These goals are crucial to implementing serious attacks such as privilege escalation and sensitive information leakage. Additionally, we design three non-standard non-trivial attack goals: `nloop` constructs a loop with instructions sufficient to ensure iteration of an arbitrary `n` times; `summation` calculates $\sum_{k=1}^{n} k$ for an arbitrary natural number n; `linkedlst` creates a singly-linked list in the memory and provides the initial node. While these attacks do not pose serious security issues, they involve more complex program logic, thus, testing the expressive power of an AEG tool in question.

To adapt BOPC to these attack goals, one should provide their descriptions in BOPC's SPLoit script language. One of the authors of this work manually wrote SPLoit scripts, and another author validated them. Any obligation invoked a discussion till an agreement has been reached. Due to the lack of arithmetic operations among registers in SPLoit syntax, such as `add ra rb`, SPLoit code for `summation` and `linkedlst` can only be generated by unrolling the loop. Consequently, BOPC cannot support general `summation` functionality without a predefined, fixed number of iterations, as specified in our attack goal. As discussed above, all real-world programs in Tab. 4 have been simplified by authors to mitigate the shortcomings of the symbolic execution engine. In the interest of a fair comparison, we have evaluate BOPC using the *same* simplified versions as DOPPLER.

Tab. 4 shows the results of the experiments. We compare the performance of DOPPLER with BOPC in achieving the attack goals discussed above. ✓represents the existence of valid assignments to specific memory space for the payload construction of the target attack goal without violating ASLR and CFI; – represents the unrealisability of the exploit in the context of the attack and program; ∞ represents the timeout of two hours; ✗ represents the non-existence of such payloads. The number next to the mark indicates the execution time in seconds. In DOPPLER, the grammar synthesis process is a one-shot execution. Users write specific attack scripts based on the generated grammar, from which generating the payload is immediate through regular expression-based syntax validation and path constraint retrieval, which is essentially a constant-time hashmap-based get operation. Hence, we omit the specific attack payload generation time in Tab. 4. The time for generating attack grammar is provided in Tab. 5.

In the first five demonstration programs (demo-N), each program is designed to contain sufficient valid statements, and the composition of these statements is flexible enough to achieve different attack goals. From Tab. 4, all of the attack goals that are designed to be realisable and can be imple-

```
1  int a, b, c, d, e; int* f=&d;  // valid variables
2  // stack buffer overflow...
3  if (a >0)
4      while (a < 10){
5          a++;
6          e += 1;}      // arithmetic
7  if (b > 0){
8      d = e;            // assignment
9      e *= e;}         // arithmetic
10 if (c > 0)
11     e += d;           // arithmetic
12 else
13     *f = e;            // store
14 if (c <10)
15     e /= 2;           // arithmetic
16 if (b < 10)
17     printf("%d",*f);  // call
18 else
19     printf("%d",e); // call
```

Fig. 5: Code snippet from demo-3.

Value	Val		integers
Valid Variables	Var		a, b, c, d, e, f
Initialisers	$Init$	::=	$Var = Val$
Valid Stmts	vs_0	::=	$e \mathrel{*}= e$
	vs_1	::=	$d = e$
	vs_2	::=	$e \mathrel{/}= 2$
	vs_3	::=	$e \mathrel{+}= d$
	vs_4	::=	$print(\text{"\%d"}, *f)$
	vs_5	::=	$*f = e$
	vs_6	::=	$print(\text{"\%d"}, e)$
	vs_7	::=	$e \mathrel{+}= 1$
Attack	$attack$	::=	$Init; ((vs_1 + (vs7 * n; vs_1))vs_0$ $((vs_5vs_4vs_6) + (vs_3((vs_4vs_6) +$ $(vs_2vs_4vs_6))))) + (vs_5vs_4) + (vs_3$ $(vs_4 + (vs_2vs_4)) + (vs_7 * n; ((vs_5$ $vs_4) + (vs_3(vs_4 + (vs_2vs_4)))))$

Fig. 6: The grammar of demo-3 from DOPPLER with a simplified regular expression on valid statements

mented by the regular language synthesised DOPPLER for each demonstration program. This shows the capability of DOPPLER to identify valid variables and statements and consolidate reachable traces into a regular grammar correctly.

In comparison, BOPC failed to produce most of the attacks for `memrd` and `memwr`, even in our demonstration programs with no more than 60 lines of code each. This result is consistent with the evaluation reported by the authors of Limbo [53]. We investigated the reason behind the failures. One of the most common outputs from BOPC is "no solution". BOPC conducts a restricted search when traversing the basic blocks for specific instructions, excluding all instructions we designed in demo programs. In addition, demonstration programs are relatively small. BOPC cannot find sufficient basic blocks with target instructions and then reports "no solution".

For the real-world programs, DOPPLER succeeds in all of the `memrd` and `memwr`, except for `sqlite` and `httpd` that failed in `memrd`, due to the lack of a statement reading from the pointer that users can overwrite. Curiously, `sqlite` achieves `nloop`, `summation`, and `linkedlst` because its valid statements contain the library function `system(zCMD)` where `zCMD` is labelled as a valid data variable in the attack grammar. Intuitively, users can make it achieve *any* attack goals, such as `summation`, by constructing the shell code in `zCMD`. Although these attacks are not implemented by directly composing valid statements from DOPPLER grammar, we still count them achievable in Tab. 4. The failures of DOPPLER in `print` in six real-world programs are caused by the lack of printing-related functions, such as `printf` and `fprtinf`, in identified reachable paths. Meanwhile, BOPC produces payload for `cherry` in `print`, `nginx` in `memrd`, gets timeout in three tasks, and explicitly fails in the rest.

In `nginx`, both DOPPLER and BOPC fail in `nloop`. However, there exists a DOP attack trace to produce loops [32] in `nginx`. BOPC can only find an infinite loop exploit by chaining a sequence of function blocks starting at function `ngx_signal_handler`, which is invoked through a function pointer `sig->handler` and setting a branch condition `ngx_time_lock` to a non-zero value. In DOPPLER, however, the control flow across the usage of function pointers defined in `ngx_signal_t` is missed by the symbolic execution engine. Thus, those potential valid statements for constructing loops are excluded by DOPPLER. In `nginx`'s attack grammar, valid statements are mainly various store and arithmetic operations from the vulnerable function `ngx_http_parse_chunked`.

We conclude the discussion on effectiveness of DOPPLER by elaborating on three specific case studies, focusing on particular vulnerable programs targeting one of three non-trivial attack goals from Tab. 4: `summation`, `print`, and `nloop`.

4.1.1 Case Study 1: demo-3

As shown in Fig. 5, demo-3 is supposed to have one `store`, one `assignment`, two `calls`, and four `arithmetic` valid statements, realising the attacks `memrd`, `memwr`, `print`, and `summation`. According to the grammar of demo-3 (*cf.* Fig. 6), DOPPLER correctly identifies the required elements. Since `a`, `b`, and `c` are control variables (*i.e.* they are used in branch conditions), the statement with control variables (line 5) is not included in the valid statement set. For the sake of the presentation, we have simplified the regular expression derived from the demo-3. With such a regular grammar, users can write the following program for calculating the `summation` of arbitrary integer values, *e.g.*, 10, as follows:

$$\{init : e = 10\}; vs_1; vs_0; vs_3; vs_2; vs_6$$

As the first statement of the program, the initialiser *init* allows users to assign initial values to data variables, corresponding to the process of corrupting vulnerable variables in

```
1  void putSDN(Printwc, fpout, outCode)
2      unsigned long int Printwc;
3      FILE *fpout; int outCode;{ // valid variables
4      static int cp=0;
5      unsigned char ibuf[1024],obuf[1024],tbuf[1024];
6      unsigned char *iptr, *tptr;
7      ...
8      if ( (Printwc>>16) == 0x8ffb ) {
9          ibuf[cp++] = (Printwc>>8)&0xff;
10         ibuf[cp++] = Printwc&0xff;
11         // buffer overflow
12     }
13     ...
14     while(*ibuf){
15         fprintf(fpout,"=?B?EUC-KR?%s?=",obuf);
16         // valid statement: call
17     }
18 }
```

Fig. 7: Code snippet from hcode where a stack buffer overflow of `ibuf` is possible if cp is larger than buffer size.

the memory space through buffer overflow at the beginning of an exploit. These variables can be arbitrary values of their types. The generated compiler will synthesise values of the remaining control variables `a`, `b`, `c`, generating a feasible payload $\{a = -1; b = 11; c = 5; d = 0; e = 10; f = 2686748\}$.

4.1.2 Case Study 2: Hcode

Hcode is a code convention library for Hangul.[5] A stack-based buffer overflow occurs in `PutSDN()` when the variable `cp` used as the index of a fixed-sized buffer `ibuf` is larger than 1024, as shown in Fig. 7. In this case, `Printwc`, `fpout`, and `outCode` are labeled as valid variables in DOPPLER. In the generated attack grammar, the function call `fprintf` at line 15 is a valid statement and can be used in the `print` attack while users gain control over the file pointer `fpout`.

4.1.3 Case Study 3: NullHttpd

Fig. 8 shows the code from *nullhttpd* where a heap buffer overflow occurs at line 4 if the value of `CoLength` is negative. Based on the structure of `CONNECTION`, we annotate eight structure members of `conn[sid]` in function `ReadPOSTData` as vulnerable variables, one of the required inputs for DOPPLER (*cf.* Sec. 3.5). To facilitate symbolic execution, we manually pruned some irrelevant program branches, such as lines 10 and 11 of printing error messages, in function `ReadPOSTData`. These simplifications *would not* affect the data flow of valid variables and the validity of the grammar in the original *nullhttpd*. Fig. 9 shows the attack grammar from DOPPLER. To make the grammar easier for users to understand, we replace complex variable names originally from the source code with shorter ones. Five valid statements consisting of pointer dereferencing operators are available to

[5] https://en.wikipedia.org/wiki/Hangul

```
1  CONNECTION *conn;
2  void ReadPOSTData(int sid) {
3      conn[sid].PostData =
4          calloc(conn[sid].dat->ConLength+1024, ...);
5      // allocate an incorrect size
6      char* pPostData = conn[sid].PostData;
7      recv(conn[sid].socket, pPostData, 1024, 0);
8      // buffer overflow
9      ...
10     if (strlen(line)==0)
11         printerror("Bad Request.", ...);
12         // pruned for simplification
13 }
```

Fig. 8: Code snippet from `nullhttpd` where a heap buffer overflow of `pPostData` is possible if `CoLength` < 0

the user, allowing them to achieve `memrd` and `memwr`. Furthermore, $vs_1 vs_0$ can be used in constructing a `nloop`.

4.2 RQ2: Efficiency of DOPPLER

DOPPLER exploit generation proceeds in two stages: (1) estimation of an attack surface by means of symbolic execution and static analysis, resulting in a synthesised attack grammar, and (2) generation of attack payload from the user's attack program written in the grammar. The first stage needs to be performed *only once*. By front loading the expensive analysis, DOPPLER amortises the cost of payload generation for all subsequent attack programs, which is now instantaneous.

Unlike DOPPLER, in BOPC [32] and Limbo [53], the entire analysis needs to be re-run for *every new attack* defined. BOPC provides limited caching functionality for analysis that is common across all attack programs, but requires significant program-specific analysis that cannot be shared. In this respect, DOPPLER has a clear advantage over BOPC once the former's grammar generation stage is completed.

Tab. 5 shows the execution times for stage (1) of DOPPLER's analysis. Subscripts in BOPC's results in Tab. 4 show the execution times of BOPC runs. Across almost all examples except `pico` and `min-dop`, DOPPLER takes longer to generate the attack grammar than any individual BOPC run. On average across all examples, stage (1) of DOPPLER

Value	Val		integers
Valid Variables	Var		$a, data$
Constant			x, b, rc, sid
Initialisers	$Init$	$::=$	$Var = Val$
Valid Statements	vs_0	$::=$	$*x = a$
	vs_1	$::=$	$**data = *a$
	vs_2	$::=$	$*rc = 0$
	vs_3	$::=$	$*x = 0$
	vs_4	$::=$	$*sid = b$
Attack	$attack$	$::=$	$Init; vs_4; vs_2; vs_3; (vs_1; vs_0) * n$

Fig. 9: The grammar fopr *nullhttpd* exploit by DOPPLER, with a simplified regular expression on valid statements

Program	Vulnerability	LOC	No. of valid statements in each type					Passive Learning				L* Algorithm			
			assign	load	store	call	arith	Σ	*N*	*P*	Time	Σ	*N*	*P*	Time
demo 1	buffer overflow	37	0	1	1	3	0	5	7	35	2	5	2	10	2
demo 2	buffer overflow	40	1	1	1	2	0	5	77	385	3	5	4	20	3
demo 3	buffer overflow	60	0	1	0	1	5	7	9	63	3	8	7	56	2
demo 4	buffer overflow	58	3	1	1	1	3	9	232	2088	10	9	8	72	10
demo 5	buffer overflow	54	0	2	0	1	4	7	189	1323	2	7	28	196	2
min-dop	heap overflow [4]	374	9	4	1	38	2	14	1048	14672	2	∞	∞	∞	∞
proftpd	CVE-2006-5815 [47]	1318	21	2	5	4	3	8	12	96	212	∞	∞	∞	∞
ghttpd	CVE-2001-0820 [23]	881	8	2	4	13	1	11	200	2200	546	∞	∞	∞	∞
sudo	CVE-2012-0809 [58]	35456	173	16	53	190	22	26	143	3718	20	26	141	22842	7184
nullhttpd	CVE-2002-1496 [43]	1715	3	2	6	28	0	5	285	1425	25	5	5	25	27
httpd	CVE-2006-3747 [28]	60796	6	0	1	3	9	6	8	48	41	19	7	133	39
nginx	CVE-2013-2028 [42]	93765	4	2	22	1	9	29	598	17342	12	44	127	5588	563
sqlite	CVE-2017-6983 [57]	195665	8	1	4	5	1	10	12	120	2	19	10	190	4
redis	CVE-2023-36824 [49]	23461	26	8	7	42	34	21	2699	56679	325	∞	∞	∞	∞
cherry	CVE-2024-22086 [17]	830	13	2	8	16	3	12	47	564	13	57	46	2565	43
pico	CVE-2024-22086 [46]	373	21	2	8	24	12	7	9	63	2	99	8	693	2
hcode	CVE-2024-34020 [26]	3679	0	0	29	12	0	21	32	672	5	38	20	760	12

Tab. 5: Statistics for the programming language artefacts synthesised by DOPPLER for the selected case studies. Σ is the size of the alphabet of a grammar's terminal symbols, *N* is the number non-terminals, and *P* is the number of productions, Time is the execution time in seconds; ∞ denotes a timeout exceeding 2 hours.

takes about 20 times as long as a single BOPC run. However, it should be noted that most of these BOPC runs did not successfully generate a payload, while DOPPLER's attack grammar had significantly more success with the selected benchmark attacks. The times for stage (2) are omitted as they are consistently less than one second (*cf.* Sec. 4.1).

The execution times for grammar generation using the L* algorithm is longer than passive learning in all benchmark programs. But the automata and grammar learned are more compact in L* algorithm. With the same size of terminal symbols, the grammar from the L* algorithm has fewer non-terminals and transition rules than passive learning. Note that L* gets a timeout of 2 hours in four programs because these programs contain large traces from symbolic execution and the algorithm is polynomial in the length of the traces.

4.3 RQ3: Language Expressivity

The expressiveness of the grammar directly affects the landscape of attacks that can be defined in the respective language. For example, in BOPC, the lack of assignment operations from register to register in SPLoit syntax makes the writing of scripts in constructing `summation` and `linkedlst` attacks very difficult (in our experience). In this section, we evaluate the expressiveness of the grammar from the perspective of valid statement types and the size of each grammar component with 17 programs. Tab. 5 shows the experiment results.

Valid Statements. A valid statement is the atomic instruction in DOPPLER grammar. In demonstration programs, DOPPLER can correctly identify valid variables and distinguish the data and control variable from the data flow propagation. For example, as mentioned in Sec. 4.1, DOPPLER correctly excludes statements `a--;` that contain a control variable `a` from the valid statement set for demo-3. In real-world examples, it is hard to build an exhaustive set of valid statements. Nevertheless, the results show that the grammar from DOPPLER contains valid statements of each type which allows for relatively expressive grammars.

Grammar Properties. We analysed the grammar size from three aspects: alphabet or terminal symbols Σ, non-terminals *N*, and production rules *P*. Tab. 5 presents the number of elements in each components. The non-terminal symbols and transaction rules describe how valid statements can be sequentially composed when constructing an attack. The size of Σ indicates the number of individual valid statement sequences users can use. More states in the grammar gives the user greater flexibility to compose valid statements together. However, larger *N* and *P* space increase the time for users to learn and understand the grammar.

Usability and Readability. As shown in Fig. 3 and Fig. 6, the generated grammars consist of mostly basic operations. It might not be immediately apparent to the user, which of those might be helpful to realise an attack. There are two potential solutions to enhance the usability and readability of the grammar: (1) augmenting valid statements with their position information would help users quickly identify the valid statements in the source code and will be easy to implement; (2) ranking valid statements in terms of their significance in constructing an attack, *e.g.*, with `execve()` in front. We leave these enhancements to the future work.

5 Limitations and Discussion

DOPPLER has been developed as a proof-of-concept prototype implementing the PLS idea for DOP attacks. As such, it comes with a number of technical limitations, most of which could be addressed with more engineering effort, which, we believe, spreads beyond the scope of this work. In this section, we outline the current limitations of DOPPLER and offer a discussion on possible improvements that could be made towards making it more scalable and user-friendly.

Interactivity. DOPPLER currently only supports one-shot exploit generation in that once source variables are assigned, they cannot be re-assigned at runtime by a user. Interactivity allows for source variables, or data pointed to by source variables to be modified by a user during the execution of the exploit at particular points in the execution. Although this feature has not yet been implemented in DOPPLER, it is relatively straightforward to extend our tool to provide valid exploits in an interactive setting. By keeping track of the current state in DOPPLER's internal DFA for a target program at the point of user interaction, and regarding this state as the new start state of the DFA, we can obtain all possible valid statements that can be correctly composed from the point of user interaction in the program. Additionally, by setting states in the DFA corresponding to subsequent interaction points as end states in the DFA, users can find sequences of valid statements that allow the user subsequent interaction opportunities.

Expressivity of synthesised grammars. As described in Sec. 3.2, a choice was made to restrict the generated grammar to regular grammars due to limitations of finite symbolic execution traces. However, there exist programs for which the correct and complete attack grammar is not regular like the one in Fig. 1 whose grammar is $(vs_1 * n); (vs_2 * n)$ which requires the expressivity of context-free grammars and push-down automata to represent. It is also possible in a similar way to come up with a programs whose attack grammar requires more expressivity than context-free grammars.

We leave exploration of contex-free grammar synthesis techniques [6] for DOP generation for future work.

Simplifications in benchmarks. As mentioned in Sec. 3, to address the limitations of KLEE symbolic execution engine, we applied the following simplifications to the real-world programs: (1) configuration file adjustments, (2) semantics-preserved rewrites, and (3) pruning of CFG irrelevant branches. During the simplification process, we manually validated that all those the changes would not affect the vulnerable data flow, vulnerable function's memory layout, or its heap space. As the result, execution traces extracted by DOPPLER and the respective payload *remained valid for the original program*. Automating these simplifications and formally proving the validity of the payload in the original programs are promising directions for future work.

```
1  int n, count0, count1;   // control variables
2  int a, b // data variables
3  char buf[1024];
4  ...
5  gets(buf); // stack buffer overflow
6
7  while(count0++ < n) {
8      printf("%d", a);  // vs1
9  }
10
11 while(count1++ < n) {
12     printf("%d", b); // vs2
13 }
```

Fig. 10: A program whose attack grammar is not regular

End-to-end exploit generation. In the exploit generation pipeline, DOPPLER requires user input in identifying memory vulnerabilities, labeling vulnerable source variables and providing annotations in the source code required by KLEE. These steps can be completed relatively mechanically following the vulnerability description in the CVE. DOPPLER is then able to automatically generate an attack grammar. Here, user input and ingenuity is required to construct a program that achieves particular attack goals, before compilation to a payload. It is possible to treat the generation of a program that respects the attack grammar and satisfies certain attack specifications as a classical program synthesis problem [25]. Using such techniques to further automate attack generation is a logical next steps following this work.

DOP mitigation techniques. Compartmentalisation [36], memory isolation schemes [52,62], and data space randomisation [7,48] are recent defence techniques against data-orient exploits. These defence techniques would affect the precision of DOPPLER when identifying valid variables and statements via detection Algorithm 1 and Algorithm 2. But they *would not* affect DOPPLER's ability to generate the payload soundly given a correct set of valid variables and statements. The algorithms of DOPPLER are indifferent to the mechanisms to bypass isolation during general data flow analysis for grammar elements and can be adapted to these defence techniques by adding stricter constraints to our learning algorithms.

What about Large Language Models? With the increasing effectiveness of Large Language Models (LLMs) at natural language text synthesis and structured reasoning tasks, it would be interesting to investigate their effectiveness at the problem of programming language synthesis. Based on some preliminary experiments, we found publicly available LLMs like GPT 3.5 to struggle with tasks like valid statement extraction and automaton learning, we believe, mostly, due to their inherent inability to soundly validate the generated results. Work in this area will have to tackle the problem of providing a more structured environment in the LLM prompt for the LLM to generate candidates, followed by their validation.

6 Related Work

Automatic synthesis of DOP-based attacks. Automatic exploit generation (AEG) [5] for memory corruptions refers to the task of hijacking control flow or manipulating data that achieves the execution of desired instructions.

Previous works on AEG of DOP attacks mainly work on data-oriented gadget identification and gadget chaining by symbolic execution or enumerated searching, including FlowStitch [29], Steroid [45], BOPC [32], and Limbo [53]. Except for BOPC, none of those tools are publicly available (*cf.* Sec. 4). While they generate DOP attack payloads under specified targets automatically, they are not always successful in finding the payload for independent attack targets [16]. In this case, the capability of the DOP attack in vulnerable software is unknown without making attempts to generate exploits for each attack target. For example, BOPC defines a Turing-complete, high-level language: SPloit Language (SPL), which allows attackers to write attack targets using SPL, executes the compiler to find an execution path, and solves real attack payloads with an SMT solver [32]. However, getting a valid gadget chain suffers from the complexity of finding a reachable path in big programs. Moreover, the BOPC compiler does not always return a result, which does not necessarily mean the absence of a satisfactory payload.

Similarly to BOPC, Limbo proposes a generic framework for finding code reuse attacks through concolic execution [53]. The basic idea of Limbo is to find the target memory state that satisfies the constraints for each attack goal. Two more recent works on Viper [64] and Einstein [33] automatically identify *syscall-guard variables* that are critical to the invocation of sys-calls for DOP attacks. However, the exploitability of *syscall-guard variables* is not guaranteed by the framework.

Language learning and program synthesis. DOPPLER relies on relatively standard automata learning algorithms to help extract potential composition rules of valid statements from accepted execution traces that are obtained from symbolic execution. Automata learning, which is equivalent to language learning, has been well developed as a sub-area of automata theory since the 1950s [3, 12, 19, 35]. Various learning algorithms are described in the literature for both regular and context-free language through the learning of DFA [59] and Pushdown Automata [39].

Defences against DOP-based attacks. Many defence techniques [7, 9, 22, 36, 48, 51, 63] have been proposed since DOP attack was first introduced [29]. Existing defence techniques like Data-Flow Integrity [15], Data-Space Randomisation (DSR) [8, 48, 63], compartmentalisation [36], data bound checking [54], and data and pointer prioritisation [2], have shown to be effective in the defence against DOP attacks. However, there is a trade off between accuracy and the efficiency. For example, pointer-based boundary checking [21, 41] suffers from high memory overhead in the analysis. Recently, Ahmed *et al.* [2] proposed a new defence technique that improves the effectiveness of the protection against DOP through the prioritisation of sensitive data. DOPPLER is not intended to help bypassing new defence schemes; instead, it automates the process of constructing a DOP exploit within the same attack as generic DOP.

Weird Machines. First posed by Bratus *et al.* [11], Weird Machines define the phenomenon of letting a program execute a sequence of instructions that could never be executed in the original program. Weird Machines provide a computational model for the behaviours of security vulnerability exploits, such as some code reuse attacks, including ROP [50], JOP [10], and DOP [29]. Recently, Paykin *et al.* [44] widened the scope of Weird Machines from exploit behaviours to *insecure compilation*. The exploit is the process of such a compilation from source language to the target language that introduces new behaviours to the program semantics, which are not feasible in the source language. While Paykin *et al.* provide a theoretical framework to formally describe actionable exploits, DOPPLER presents its working implementation.

7 Conclusion and Future Work

In this work, we have studied the challenge of generating code reuse attacks. Our key new insight was to consider it from the perspective of the novel *programming language synthesis* idea, deriving grammars of languages for expressing (possibly infinite) sets of feasible attacks, rather than synthesising the attacks on a case-by-case basis. Our approach is first to provide formal *soundness* guarantees *wrt.* realisability of attack synthesis tasks: any attack expressible in terms of the synthesised grammar can be realised with a suitable payload, which our tool DOPPLER is able to find automatically. A promising extension of our results would be to improve the quality of the generated attack grammars by synthesising high-level programming constructs with the help of formal techniques for reasoning about unrealisability [31, 34].

An important direction for the future work is to address the complementary *completeness* guarantees, stating that *every* possible attack is captured by the synthesised language. The completeness of an attack language synthesis is critical to assess *to which extent* a given vulnerable program is exploitable, thus characterising the set of code reuse attacks that *cannot* be implemented on top of it. At the moment, our approach cannot guarantee completeness due to the inherent unsoundness of the underlying symbolic execution engine [14], which cannot provably identify the set of *all* reachable states. In the future, we will consider a combination of symbolic execution with sound abstract interpretation [18] as an approach to prove *completeness* of attack language synthesis (*i.e.*, that the synthesised grammar covers *all* possible attacks), while possibly sacrificing its soundness, *i.e.*, making some attacks expressible in the synthesised language unrealisable.

Acknowledgements

We thank Prateek Saxena, Julien Vanegue, Naipeng Dong, and Jin Song Dong for their comments. We also thank the anonymous reviewers of USENIX Security '25 and our shepherd for their feedback. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001, MoE Tier 1 grant T1 251RES2108 "Automated Proof Evolution for Verified Software Systems", and by Sui Academic Research Award.

Open Science Statement

The software artefact accompanying this paper is available online [37]. The artefact contains the source code of DOPPLER and the build/execution harness required to reproduce the experiments described in Sec. 4.

Ethics Considerations

Our work does not disclose any previously unknown vulnerabilities, neither does it describe new forms of attacks. That said, we do introduce new ideas for faster generation of code reuse attacks by exploiting documented vulnerabilities. We believe, the benefits of publishing these ideas, along with the open-source implementation of our tool and a clear characterisation of its practical limitations, outweigh the potential harm they might cause when used with malicious intent.

In particular, we believe that our formal characterisation of the *soundness* aspect of DOP-based code reuse attacks will make it possible for the potential users of the known vulnerable programs to quantify upfront what attacks *can* be executed reliably—an important guarantee that has not been considered by prior works on code reuse attack generation. Furthermore, our conceptual methodology opens a possibility to provably claim that certain attacks *cannot* be implemented—a direction we chart as promising future work.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), 2009.

[2] Salman Ahmed, Hans Liljestrand, Hani Jamjoom, Matthew Hicks, N. Asokan, and Danfeng Yao. Not all data are created equal: Data and pointer prioritization for scalable protection against data-oriented attacks. In *USENIX Security Symposium*. USENIX Association, 2023.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2), 1987.

[4] Miguel A. Arroyo. Minimal data-oriented programming vulnerability + exploits. https://github.com/mayanez/min-dop, 2020. Last accessed: 03/09/2024.

[5] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2), 2014.

[6] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *PLDI*. ACM, 2017.

[7] Brian Belleville, Hyungon Moon, Jangseop Shin, Dongil Hwang, Joseph M. Nash, Seonhwa Jung, Yeoul Na, Stijn Volckaert, Per Larsen, Yunheung Paek, and Michael Franz. Hardware Assisted Randomization of Data. In *RAID*, volume 11050 of *LNCS*. Springer, 2018.

[8] Sandeep Bhatkar and R. Sekar. Data space randomization. In Diego Zamboni, editor, *DIMVA*, volume 5137 of *LNCS*. Springer, 2008.

[9] Tyler K. Bletsch, Xuxian Jiang, and Vincent W. Freeh. Mitigating code-reuse attacks with control-flow locking. In *ACSAC*. ACM, 2011.

[10] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *AsiaCCS*. ACM, 2011.

[11] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. Exploit programming: From buffer overflows to "weird machines" and theory of computation. *login Usenix Mag.*, 36(6), 2011.

[12] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4), 1964.

[13] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1), 2017.

[14] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. USENIX Association, 2008.

[15] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-Flow Integrity. In *OSDI*. USENIX Association, 2006.

[16] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N Asokan, and Danfeng Yao. Exploitation techniques for data-oriented attacks with existing and potential defense approaches. *ACM Transactions on Privacy and Security (TOPS)*, 24(4), 2021.

[17] Cherry CVE-2024-22086. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-22086, 2024. Last accessed: 03/09/2024.

[18] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*. ACM, 1977.

[19] Crispan Cowan. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*. USENIX Association, 1998.

[20] Data execution prevention. https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention. Last accessed on September 3, 2024.

[21] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: architectural support for spatial safety of the C programming language. In *ASPLOS*. ACM, 2008.

[22] Irene Díez-Franco and Igor Santos. Data Is Flowing in the Wind: A Review of Data-Flow Integrity Methods to Overcome Non-Control-Data Attacks. In *SOCO'16-CISIS'16-ICEUTE'16*, volume 527 of *Advances in Intelligent Systems and Computing*, 2016.

[23] GHttpd CVE-2001-0820. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0820, 2001. Last accessed: 03/09/2024.

[24] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *PLDI*. ACM, 1991.

[25] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2), 2017.

[26] hcode CVE-2024-34020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-34020, 2024. Last accessed: 03/09/2024.

[27] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

[28] Apache Httpd CVE-2006-3747. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3747, 2006. Last accessed: 10/01/2025.

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

[30] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *S&P*. IEEE Computer Society, 2016.

[31] Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas W. Reps. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In *PLDI*. ACM, 2020.

[32] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In *CCS*. ACM, 2018.

[33] Brian Johannesmeyer, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical data-only attack generation. In *USENIX Security Symposium*. USENIX Association, 2024.

[34] Jinwoo Kim, Loris D'Antoni, and Thomas W. Reps. Unrealizability logic. *Proc. ACM Program. Lang.*, 7(POPL), 2023.

[35] S.C. Kleene. Representation of Events in Nerve Nets and Finite Automata. *Annals of Mathematics Studies*, 34, 1956.

[36] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. SoK: Software Compartmentalization. *CoRR*, abs/2410.08434, 2024.

[37] Yuxi Ling, Gokul Rajiv, Kiran Gopinathan, and Ilya Sergey. Sound and Efficient Generation of Data-Oriented Exploits via Programming Language Synthesis (Artifact), 2025. Available at https://doi.org/10.5281/zenodo.14718582.

[38] Mark Moeller, Thomas Wiener, Alaia Solko-Breslin, Caleb Koch, Nate Foster, and Alexandra Silva. Automata learning with an incomplete teacher. In *ECOOP*, volume 263 of *LIPIcs*, pages 21:1–21:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[39] David E Muller and Paul E Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 37, 1985.

[40] Kevin P. Murphy. Passively learning finite automata. Technical Report 1996-04-017, SFI, 1996.

[41] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *PLDI*. ACM, 2009.

[42] Nginx CVE-2013-2028. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028, 2013. Last accessed: 10/01/2025.

[43] nullhttpd CVE-2002-1496. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-1496, 2002. Last accessed: 03/09/2024.

[44] Jennifer Paykin, Eric Mertens, Mark Tullsen, Luke Maurer, Benoît Razet, Alexander Bakst, and Scott Moore. Weird machines as insecure compilation. *CoRR*, abs/1911.00157, 2019.

[45] Jannik Pewny, Philipp Koppe, and Thorsten Holz. STEROIDS for doped applications: A compiler for automated data-oriented programming. In *EuroS&P*. IEEE, 2019.

[46] Pico CVE-2024-22086. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-22086, 2024. Last accessed: 03/09/2024.

[47] ProFTPD CVE-2006-5815. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5815, 2006. Last accessed: 03/09/2024.

[48] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. CoDaRR: Continuous Data Space Randomization against Data-Only Attack. In *AsiaCCS*. ACM, 2020.

[49] Redis CVE-2023-36824. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-36824, 2023. Last accessed: 10/01/2025.

[50] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1), 2012.

[51] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z. Snow, and Michalis Polychronakis. Revisiting browser security in the modern era: New data-only attacks and defenses. In *EuroS&P*. IEEE, 2017.

[52] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *USENIX Security Symposium*. USENIX Association, 2022.

[53] Edward J. Schwartz, Cory F. Cohen, Jeffrey Gennari, and Stephanie Schwartz. A generic technique for automatically finding defense-aware code reuse attacks. In *CCS*. ACM, 2020.

[54] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*. USENIX Association, 2012.

[55] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS*. ACM, 2004.

[56] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *S&P*. IEEE Computer Society, 2013.

[57] Sqlite CVE-2017-6983. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6983, 2017. Last accessed: 10/01/2025.

[58] sudo CVE-2012-0809. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0809, 2012. Last accessed: 03/09/2024.

[59] M. A. L. Thathachar and P. S. Sastry. Varieties of learning automata: an overview. *IEEE Trans. Syst. Man Cybern. Part B*, 32(6):711–722, 2002.

[60] Timothy Tsai and Navjot Singh. Libsafe 2.0: Detection of format string vulnerability exploits (white paper). *Avaya Labs*, 2001.

[61] Timothy K. Tsai and Navjot Singh. Libsafe: Transparent system-wide protection against buffer overflow attacks, 2002.

[62] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security Symposium*. USENIX Association, 2019.

[63] Stijn Volckaert. Randomization-based defenses against data-oriented attacks. In *MTD@CCS 2021: Proceedings of the 8th ACM Workshop on Moving Target Defense*. ACM, 2021.

[64] Hengkai Ye, Song Liu, Zhechang Zhang, and Hong Hu. VIPER: Spotting Syscall-Guard Variables for Data-Only Attacks. In *USENIX Security Symposium*. USENIX Association, 2023.