Searching for Software Diversity

Attaining Artificial Diversity through Program Synthesis

Gilmore R. Lundquist
The University of Texas at Dallas
gilmore.lundquist@utdallas.edu

Vishwath Mohan
Google
vishwath.mohan@gmail.com

Kevin W. Hamlen
The University of Texas at Dallas
hamlen@utdallas.edu

ABSTRACT

A means of attaining richer, more comprehensive forms of software diversity on a mass scale is proposed through leveraging and repurposing a closely related, yet heretofore untapped, line of computer science research—automatic program synthesis. It is argued that the search-based methodologies presently used for obtaining implementations from specifications can be broadened relatively easily to a search for many candidate solutions, potentially diversifying the software monoculture. Small-scale experiments using the Rosette synthesis tool offer preliminary support for this proposed approach. But the possible rewards are not without danger: It is argued that the same approach can power a dangerous new level of sophistication for malware mutation and reactively adaptive software threats.

CCS Concepts

•**Security and privacy** → **Software security engineering;** •**Software and its engineering** → *Source code generation; Automatic programming;* Search-based software engineering;

Keywords

Artificial diversity; program synthesis; security

1. INTRODUCTION

For almost a quarter century—since Cohen's seminal 1993 article on program evolution [13]—the brittleness of *software monoculture* has been recognized by the computer security community as a major root of cyber insecurity. In contrast to most biological systems, whose chaotic diversity engenders communal resilience against many threats, such as disease, environmental change, or species introduction; software systems tend towards uniformity, championing ideals such as code reuse, computation replication, and resource sharing. Attacks that succeed against one of these shared elements therefore too often succeed against the entire community, threatening to eradicate entire software species in a single blow.

Cohen recognized the value of diversity in part because he had already observed its value for cyber offense. His 1986 dissertation [12] prophetically observes that viral evolution ultimately dooms static approaches to antivirus detection due to the limits of computability (a verdict that the antivirus industry recently recapitulated 28 years later under the mantra "antivirus is dead" [80]). Thus, like competing biological communities, the most diverse software community tends to have the greatest survivability.

Over the following decades, scientists responded to these observations by advocating what might be termed *artificial micro-diversity*—the perturbation and randomization of low-level computer code to produce syntactically different but semantically equivalent code. For example, Forrest's influential 1997 paper [19] promoted dead code introduction, basic block reordering, and memory layout randomization as promising implementation strategies for defensive diversity. Such perturbations can probabilistically defeat many low-level exploits, such as buffer overflows, and have the advantage of being potentially applicable to existing software products without requiring redevelopment.

But large classes of attacks remain immune to micro-diversity. For example, cross-site scripting (XSS)—one of the historically largest software vulnerability classes—is generally attributable to a failure to suitably sanitize untrusted inputs flowing to trusted sinks. Any diversification strategy that diligently preserves program semantics is likely to preserve the sanitization lapse,[1] leading once again to universal vulnerability of the community. Thus, true software diversity remains an elusive challenge in many respects.

Concurrently and largely independently, the programming languages and compilers community has devoted a large body of research to a different but closely related problem: automatic *program synthesis*. Synthesis work is motivated by the observation that humans tend to be much better at describing what computer programs should do than writing efficient and correct computer programs that do it. The goal of synthesis is therefore typically defined as, "the systematic derivation of a program from a given specification" [48] (cf., [47]).

In this paper, we propose retargeting advances in program synthesis to the artificial diversity problem by broadening this goal to "the systematic derivation of *a diversity of programs* from a given specification." All synthesis research is rooted in search (because there are generally an infini-

[1]Some XSS cases can be addressed by blocking other aspects of program behavior than input sanitization (e.g., [21]).

NSPW '16 September 26-29, 2016, Granby, CO, USA

ACM ISBN 978-1-4503-4813-3/16/09.

DOI: http://dx.doi.org/10.1145/3011883.3011891

tude of programs that satisfy any given specification), so broadening the search to multiple programs seems both natural and potentially feasible. Synthesis algorithms tend to discard "poor" (e.g., less efficient but still correct) search candidates as probably undesirable to users; but from a diversity standpoint, the synthesist's dross may be the defender's silver. In particular, we consider whether a broadened form of program synthesis may be a feasible avenue toward more *macro-diverse* software communities. Dually, we consider program synthesis as a potentially dangerous methodology for aggressively metamorphic malware offense.

Toward exploring these questions, we survey pertinent research on program synthesis and artificial diversity, and conduct some preliminary experiments on marrying these heretofore disparate technologies. The outcomes of these experiments suggest that diversity is obtainable via program synthesis and has the potential to replace or outperform existing diversity techniques currently in use.

The remainder of the paper proceeds as follows. In Section 2, we motivate the need for program diversity for both defensive and offensive security by examining recently published surveys of the literature [3, 39]. Section 3 correspondingly draws upon surveys of the program synthesis literature [22] to summarize modern synthesis methods and common applications. Section 4 describes our initial attempts at attaining program diversity using synthesis. Section 5 outlines discussions that occurred at NSPW. Section 6 proposes future work in this area. Section 7 concludes.

2. ARTIFICIAL DIVERSITY

Artificial diversity refers to the creation of semantically equivalent but syntactically different variants of a given program for security and reliability purposes. The idea was initially proposed over two decades ago [13] as a means of fault-tolerance for mission critical systems. It easy to see how nature inspires such a proposal. Biological diversity ensures population-wide resilience against a specific disease or virus, and forms a critical factor in the survivability of a species (and even the ecosystem as a whole) [30, 46, 50].

Although the term "diversity" has traditionally been applied to defense, we show later in this section that some popular methods employed by malware to evade detection are conceptually identical to defensive diversity.

Our focus here is on examining diversity techniques and methodologies; for a broader discussion of the motivations of diversity in the context of full system design, we refer the reader to related works [5, 6, 60, 70, 73].

2.1 Defensive Diversity

Software (and hardware) is often intentionally engineered to be homogenous. Homogeneity simplifies the effort involved in developing, distributing, maintaining, and updating a product, making it the commercially advantageous choice.

However, this homogeneity also lowers the bar for exploitation. Malicious actors who obtain and analyze any instance of a product derive conclusions that reliably apply to all instances. Vulnerabilities they find can be therefore be developed into exploits that threaten all copies of the product.

Defensive software diversity aims to introduce uncertainty between different versions of a software product, requiring attackers to obtain precise knowledge of the target versions they wish to exploit. This means that a vulnerability found in one instance of the product does not necessarily exist in other copies, forcing attackers to tailor exploits to the specific copy of the product they wish to target. This significantly raises the bar for both targeted and mass scale attacks.

Various types of diversity have been proposed that operate at different granularities[2] and at different points in the software development and deployment lifecycle.[3] Larsen et al. [39] categorize different diversity schemes based on the level at which they operate, as well as when in the lifecycle they are utilized. A partial, abbreviated listing of these is given below.

2.1.1 Diversity by Level

1. *Instruction level* approaches attempt to replace instructions or instruction sequences with semantically equivalent, but different instructions (or instruction sequences). Alternately, some approaches also limit themselves to reordering existing instructions or instruction sequences.

2. *Basic block level* approaches to diversity typically permute the order of basic blocks [79], or obfuscate the control flow graph [14, 44].

3. *Function level* techniques include in-lining function code at call sites, extracting code fragments into new functions and replacing them with calls in the original code, randomizing the order of parameters to functions, and randomizing stack layouts.

4. *Program level* diversity re-orders the layout of functions or function tables (e.g., the GOT and PLT structures in ELF binaries), randomizes the base addresses of binary sections (e.g., ASLR), or reencodes the program in a different format [26, 35, 66].

2.1.2 Diversity by When

1. *Design and implementation* stage diversity entails implementing multiple versions of a component using different algorithms, programming languages, and/or personnel. This can drastically reduce the chance that a vulnerability found in one version of a component is also applicable to other versions.

   Diversity at this stage is a costly process, and thus not widely adopted. It is most often used as a means of introducing fault-tolerance [1, 62] in select domains like aerospace, where reliability is a critical concern.

2. *Compilation and linking* stage techniques are largely automatic, and proposed more for security than for fault-tolerance. Compilation-based approaches rely on one or more compiler passes that perform the actual diversification, while linking-based approaches take advantage of debugging information from the compiler's output to perform diversification just before linking separately compiled modules.

[2]For a discussion of the special challenges incurred from deploying defensive diversity in embedded systems, see McLaughlin et al. [51].

[3]For a discussion of deploying defensive diversity at installation time from a large-scale ("App-store" style) deployment system, see Franz [20].

3. *Post-installation* approaches statically rewrite binary programs after they are installed, but before they are executed. For the most part, this rewriting does not perform the actual diversification, but instead prepares the binary to be diversified when it is loaded. Post-installation approaches tend to focus mainly on randomizing the relative locations of instructions [26] or basic-blocks [79], and do not attempt to replace instruction sequences with semantically equivalent substitutes.

4. *Load-time* approaches modify the in-memory representation of a process before it is executed. Some of these tend to be secondary stages of post-installation approaches, but there also exist purely load-time diversification techniques [17].

In general, we observe that implementation and design level techniques tend to produce changes at the highest conceptual levels—at a whole program or algorithmic level—while the remaining pre- and post-installation based approaches tend to work between the instruction and function levels. This is offset by the increased cost and difficulty of scaling this approach.

Additionally, most automated approaches focus on permuting or randomizing code positions at different granularities—from individual instructions to entire functions. Fewer techniques attempt instruction (or instruction sequence) replacement.

2.2 Offensive Diversity

Program diversity is not just useful as a defensive technique; it is one of the fundamental methods employed by malware to evade detection.

End-user detection for malware tends to rely heavily on various forms of static and syntactic analysis, in addition to dynamic monitoring, because of the higher costs associated with dynamic and behavioral analysis [37, 38, 43, 56, 67]. This is reflected in the evasion techniques employed by malware, which can be broadly split into the following categories:

1. *Oligomorphic malware* uses simple invertible instructions, such as exclusive-or, to transform malicious code bytes and hide distinguishing syntactic features. Before the code executes, the original contents are recovered by applying the inverse transformation to the obfuscated payload.

2. *Polymorphism* is an evolution of the idea behind oligomorphy. In lieu of simple operations, polymorphic malware encrypts the payload, or applies a similarly one-way function, leaving only a small decryption routine behind. When the malware is loaded, the decryption routine unpacks the payload and transfers control to it.

3. *Virtualization-based obfuscation* expresses payloads in terms of a bytecode language tailored for a custom embedded virtual machine (VM). The VM executes the payload by interpreting the bytecode at runtime. By mutating the bytecode language (and the VM) on each propagation, a high degree of diversity can be maintained.

4. *Metamorphic malware* replaces malicious code sequences with semantically identical code during propagation. It accomplishes this using a metamorphic engine that introspects its own binary programming and performs modification passes. These passes replace instruction sequences with structurally different but semantically equivalent sequences, resulting in high diversity without the telltale entropy produced by polymorphism, or the potentially distinguishing VM of virtualization-based malware.

As evident from their descriptions, oligomorphic and polymorphic malware obfuscate themselves by hiding the contents of the payload from analyses syntactically, while malware that employs virtualization-based obfuscation or metamorphism aims to thwart detection using a more semantic style of diversity.

Of these approaches, metamorphism is potentially the hardest to detect. Oligomorphic and polymorphically encoded payloads exhibit identifiable statistical properties, such as entropy [45] and byte frequency [77, 78], that are unique from most benign code, and which can be used to detect such malware in the wild. Virtualization based obfuscation effectively shifts the malware's obfuscation burden to hiding the in-lined, custom VM, and therefore has the potential to be statically detectable using carefully selected heuristics that exploit this weakness.

Because metamorphism introduces mutations by replacing code sequences with equivalent plaintext code, it does not typically exhibit statistical properties that distinguish it from non-encrypted benign software. Additionally, there are no complicated interpreters or JIT-compilers in-lined into metamorphic malware, allowing it to bypass heuristics that would detect VM-based approaches.

Most metamorphic engines use a bottom-up approach that consists of disassembling native code to some intermediate representation, adding diversity by modifying the intermediate code in functionally equivalent ways, and finally converting it back into (now mutated) native code. They generally perform some combination of the following five phases to introduce diversity [58]:

1. *Garbage insertion* adds unreachable code to the original code.

2. *Code substitution* replaces instructions or instruction sequences with semantically identical versions.

3. *Code insertion* adds dead ineffectual code in between the actual malicious payload.

4. *Register swapping* reallocates the registers that instructions use.

5. *Control flow scrambling* uses branch instructions to reorder basic blocks or functions.

Interestingly, we observe that many of these techniques are identical to the instruction level, basic-block level and function level transformations used for defensive diversity.

Offensive research has also explored the merits of a top-down approach to malware obfuscation, as demonstrated by the Frankenstein system [54]. Frankenstein starts with a high-level representation of the payload logic, concealed within its data sections. When deployed on a target system, it disassembles benign binaries and constructs a database of *gadgets*—instruction sequences that perform useful semantic

tasks—and uses the process of unification from logic programming to find a sequence of gadgets that is semantically equivalent to the high-level payload description. The use of unification imbues Frankenstein with the ability to generate mutants that more closely exhibit the statistical properties of benign binaries unique to a given victim machine or network.

A comparison of defensive versus offensive diversification strategies reveals at least two major differences between the two: First, offensive diversity is more likely to trigger at propagation time than at any of the stages described in Section 2.1. This makes sense given that the primary purpose of diversity for malware is to evade static detection during transit and at rest, while the motivation for defensive diversity is to eliminate common vulnerabilities or exploitation paths.

Second, the adoption of defensive diversity techniques is far more constrained by factors such as runtime performance and maintainability, which are not as high priorities for many malware authors. For example, most malware can afford to run with higher overhead if it means increased stealth. These differences influence how and what kinds of program synthesis can be used for both purposes.

3. PROGRAM SYNTHESIS

Program synthesis [2,4,8,16,22–24,27,29,31–34,40–42,47–49,52,61,68,69,71,72,75,76] tackles the problem of systematically deriving or discovering a program from a specification [22,47,48]. The synthesized program therefore has some formal guarantee of correctness with respect to the given specification; however, specifications are usually incomplete from a security perspective. For example, a specification for a string sorting computation would typically prescribe orderedness of output and one-to-one correspondance of input to output elements, but would not typically enumerate all standard library functions that have vulnerabilities, and that therefore *must not* be employed in any synthesized implementation. This is especially the case if some vulnerabilities are as yet unknown to defenders. Hence, synthesis does not guarantee security even if the synthesis algorithm is correct.

While compilers and other translators play an important role in the synthesis problem space, a distinguishing characteristic of synthesis over compiler construction is the former's heavy reliance on search techniques to find solutions. These searches can wander farther afield from the programmer's original input specification than traditional syntax-directed compilation, allowing specifications to be less comprehensive than traditional source code programs, and leading to solutions farther removed from specfications than object code is removed from source code. It can also provide new algorithmic insights in the search results [22].

Program synthesis' goal of producing a singular program seems to be prevalent throughout the history of the literature. For this work, we propose generalizing the goal of generating "a program" to generating "a diversity of programs". By so doing, we hope to capitalize on the capacity of search to explore a much larger and more diverse space of solutions than traditional compilers. We feel this could potentially offer great benefits to cyber security, toward more effective artificial diversity for cyber defense, and more potent software polymorphism for cyber offense.

3.1 Synthesis Methods and Applications

Gulwani [22] separates program synthesis methods into three dimensions: user intent (i.e., the format of the input specification), search space, and search technique.

Input specifications are often based on logic (e.g., [23,29,48,61,72]) or proofs (e.g., [47]). Other common choices are natural language [24,42,61], which can be converted via natural language processing into a logical specification; example inputs and outputs [2,41,52], including program traces [4,16,40]; and example programs or reference implementations, as in the case of sketching (discussed in Section 3.2) and synthesizing program inverses [71].

Of these, we speculate that program traces and program sketching are perhaps the most promising and relevant approaches for security applications. Traces have natural links to security because security policies are often specified in terms of valid program traces [9]. Alternatively, program sketches more closely resemble traditional computer programs, and might therefore be a more feasible approach to diversifying existing code. Our preliminary experiments reported in Section 4 therefore adopt a sketching approach.

A synthesis engine typically converts a given input specification into constraints for some search method, which is then applied to find a result program from the search space. Boolean satisfiability (SAT) solvers and satisfiability modulo theories (SMT) solvers are standard vehicles for search, but many other search algorithms are also used (e.g., A* goal directed search, version space algebras [2, 41, 52], or brute force search [8,34,49,52]). SAT/SMT solvers are sometimes combined with other search algorithms.

3.2 Program Sketching

Of the aspects of synthesis systems mentioned in Section 3.1, the most important to the user of the system is the choice of input specification. In addition to being sufficiently general (many synthesis systems are domain specific), for our experiments we wanted a system whose input specification was easy to work with and allowed control over how much of the program to synthesize. In particular, we wanted the ability to influence whether and how much loops and other control flow constructs are synthesized, since these are often harder for synthesis engines to handle.

Toward this end, we focused on systems that take a program sketch [27,31,68,69,75] as input. A program sketch is a program with holes to be filled in by the synthesis system. The sketch both constrains the search space to those programs that match the sketch, and also provides a correctness specification in the form of assertions defining correct program behavior. Assertions and holes can then be converted into an integer constraint problem to be solved by an SMT solver.

For example, consider the code listed below (from [68], written in the SKETCH language). The program contains an integer hole, denoted by '??'. The assertion formally specifies the desired behavior of the program.

```
void main(int x){
   int y = x * ??;
   assert y == x + x;
}
```

Synthesis is performed by converting holes into parameters of the program, and then partially evaluating the sketch using the results from the solver as static inputs, which fill in the holes. No assertions may fail for any input to the program. The synthesized result of the example above is identical to the input sketch, but with the hole replaced with

the constant '2'. In this case there is only one solution, but in general there may be many possible valid completions.

Parts of the sketch outside of the holes must remain unchanged in the output, while the synthesis engine may fill in the holes as it sees fit. This allows the programmer to provide a template for the structure of the program, and to constrain the search space as much or as little as desired. Loops are typically provided outside of holes in the sketch to avoid the intractability mentioned above.

More complex code may be synthesized by providing a grammar from which possible completions may be generated. Alternatives in the grammar are converted automatically into integer or boolean holes representing the choices made and then solved as before.

3.3 Diversity in Synthesis

It seems clear that diversity is naturally present, if not prevalent, in the problem of synthesizing code; in fact, oftentimes designers of synthesis systems devote great effort toward hiding less-than-optimal solutions from users when selecting a final solution. The presumption is, of course, that users want to see only the best solution discovered by the algorithm.

To our knowledge, no prior work has applied program synthesis to the problem of artificial diversity. While security is occasionally mentioned in the synthesis literature, it tends to be limited to reverse-engineering and malware de-obfuscation applications [33, 40] and bug fixes [57]. Such applications still aim to produce a single program as output, and avoid methods that yield a plethora of solutions.

Evolutionary computation (EC) has recently been applied to fix security bugs in router firmware [64]. EC is not strictly the same as program synthesis, however there are similarities. While EC does create a diversity of mutants internally as part of the process of evolution, none of these candidate programs are viable for use since the algorithm terminates as soon as it finds one able to pass its regression tests. We believe that this and other non-synthesis search algorithms which operate on programs may also be able to be adapted as proposed in this paper to provide a diversity of (viable) outputs.

Multivariant or *N-variant systems* detect intrusions or faults by running a monitored diversity of semantically equivalent programs simultaneously [15]. Diversity in these systems is typically limited to primarily micro-diverse techniques (see Section 2.1.1). Richer sources of diversity, including our proposed approach, can therefore potentially benefit such approaches by supplying an automated source of more macro-diverse, yet semantically equivalent variants—perhaps with different intrusion detection capabilities.

4. PRELIMINARY EXPERIMENTS

4.1 Rosette

For our preliminary experiments in applying synthesis to diversity, we chose the Rosette language [74, 75, 76]. Rosette provides synthesis, verification, and symbolic execution via solver-aided constructs; symbolic models are converted to integer constraint problems and solved with an SMT solver of the user's choice—Microsoft's Z3 [55] by default.

Rosette is embedded within Racket [18], a functional language that is a successor to Scheme and Lisp. The availability of Racket within Rosette is particularly useful for our investigation because it provides many tools for extending the language's syntax and functionality, giving the developer the ability to create new programming languages or Domain Specific Languages (DSLs). The Rosette system provides two language dialects:

- a *safe* dialect (`#lang rosette/safe`), providing some of Racket's features, but omitting those which might interfere with the solver; and
- an *unsafe* dialect (`#lang rosette`), providing all of Racket's features.

These languages are designed to allow users to create their own DSLs with synthesis and verification capabilities built in.

Rosette uses program sketching (see Section 3.2) to synthesize code. A library is provided that allows the user to specify a recursive grammar, or *synthax* (a portmanteau for "synthesizable syntax"). Synthax grammars are converted into binary symbolic variables that encode the grammar alternatives chosen by the synthesis engine.

4.2 A Simple Synthesis Example

As an example,[4] consider the following problem: given two machine registers (r_1 and r_2) and a pool of small program fragments, which we term *gadgets* [65], construct a sequence of gadgets that stores the value 4 in register r_1. While our use of the term *gadget* is inspired by offensive security (*return-oriented programming* in particular), we use the term here in a more general sense: A gadget is any code fragment with known semantics.

First, we use Rosette's `bitvector` type to define a subtype with a given fixed precision, namely 8 bits. This new type will be used to model 8-bit registers:

```
#lang rosette/safe

(define bitwidth 8)
(define bv8? (bitvector bitwidth))
(define bv8-const (lambda (v) (bv v bitwidth)))
```

The `#lang` directive tells the Racket interpreter which language should be used to parse this module's source code. The `bv8?` predicate returns true (`#t`) if the argument is of our defined type, and `bv8-const` constructs a constant value in our defined type.

Next, we define symbolic variables to represent the registers. Here we use Rosette's symbolic execution, which lets us symbolically compute constraints on the possible values of each register after each step. Function `define-symbolic` creates a fresh symbolic value for each name given, with the final argument determining the type. Since each gadget destructively updates the value of the destination register, we save a copy of the unmodified symbolic value immediately after creation for later use. We also provide a `reset` function to restore the value of the registers to their original (unmodified) symbolic value.

```
; model 8-bit registers r1 and r2
(define-symbolic r1 r2 bv8?)
```

[4]Our example code can be downloaded from https://www.utdallas.edu/~hamlen/projects.html.

Table 1: Gadgets and their computations

Gadget	Operation
G1	$r_1 \leftarrow r_1 \times 2$
G2	$r_2 \leftarrow r_2 + 1$
G3	$r_1 \leftarrow r_1 - 1$
G4	$r_1 \leftarrow 0$
G5	$r_1 \leftarrow 6$
G6	$r_2 \leftarrow 1$
G7	$r_1 \leftarrow r_2$
G8	$r_2 \leftarrow r_1$

```
; gadgets are side-effectful;
;  use (reset) to clear them.
(define r1-symb r1)
(define r2-symb r2)
(define (reset)
  (set! r1 r1-symb)
  (set! r2 r2-symb))
```

Next, we define our gadget pool, and a helper function to execute a sequence of them. Here we use the bitvector operators provided by Rosette, whose names start with the prefix 'bv'. For example, `bvshl` performs a shift-left on bitvectors. While the gadgets presented here are each equivalent to a single instruction, this need not be the case in general. (We only show the implementation of a few gadgets; a full list is given in Table 1):

```
; G1:  r1 *= 2
(define (g1)
  (set! r1 (bvshl r1 (bv8-const 1))))

; G2:  r2 += 1
(define (g2)
  (set! r2 (bvadd r2 (bv8-const 1))))

        ⋮

; G5:  r1 <- 6
(define (g5) (set! r1 (bv8-const 6)))

        ⋮

; G8:  r2 <- r1
(define (g8) (set! r2 r1))

; given a gadget list, execute each in sequence.
(define (run-code code)
  (map (lambda (g) (g)) code))
```

Here, `(run-code` ⟨*gadget-list*⟩`)` takes a list of gadgets and evaluates each gadget as a function, returning a list of results. This has the useful property of executing the side-effects of each gadget in order.

Next, we define our desired goal for the synthesized program. We do this by asserting the desired properties using Rosette's `assert` function. This has two uses: when called with a proposition that uses concrete values, `assert` acts like an assert in a typical imperative programming language—if the proposition is false, it causes a runtime failure; otherwise execution continues with no effect. If called on propositions containing symbolic values, Rosette adds the (symbolic) proposition to a list of constraints to be checked next time the solver is exercised.

```
; GOAL: code that produces the result r1 = 4
(define (assert-correct)
    (assert (equal? r1 (bv8-const 4))))

(define (code-correct code)
  (reset)
  (run-code code)
  (assert-correct))
```

Function `code-correct` prescribes that a given gadget sequence is correct if, when executed after a reset (i.e., with any possible starting values in the registers), the desired property is true (namely that register r_1 contains the value 4).

With this infrastructure in place, we're finally ready to synthesize some code. First, we use `require` to include a library for defining synthax. We then define `gadget-sequence`, a synthax grammar for generating a list of gadgets of given depth. The Rosette function `choose` creates a hole to be filled with any of the given expression alternatives. This definition encodes the following grammar:

gadgetSeq ::= (`list` *gadget*) | (`cons` *gadget gadgetSeq*)
gadget ::= `g1` | `g2` | `g3` | `g4` | `g5` | `g6` | `g7` | `g8`

We then define[5] `synthesized-sequence`, a sketch with a `gadget-sequence` hole. The sequence depth is passed as an argument.[6] Holes in the sketch determine the choice of gadgets and therefore the permutation of the sequence.

```
(require rosette/lib/synthax)

; a grammar for gadget lists
(define-synthax (gadget-sequence depth)
  #:base (list [choose g1 g2 g3 g4 g5 g6 g7 g8])
  #:else (cons [choose g1 g2 g3 g4 g5 g6 g7 g8]
           (gadget-sequence (- depth 1))))

; a sketch with a hole, to be filled
;   with gadget sequences of length 3
(define synthesized-sequence (gadget-sequence 2))

; synthesize a correct gadget sequence:
; find a concrete model for the symbolic variables
;   that satisfies all of our assertions
(define soln
 (synthesize
  #:forall (list r1 r2)
  #:guarantee (code-correct synthesized-sequence)))

(if (sat? soln)
    (print-forms soln)
    (display "Unsatisfiable"))
```

We then call `synthesize` to complete the sketch, asserting that our predicate `code-correct` must be true for the generated sequence. The result of `synthesize` defines a model assigning values to all symbolic variables except those given in the `#:forall` list. If the model is satisfiable, then the values in the model are used to complete the sketch, which synthesizes a particular gadget sequence.

[5]At the time of this writing, sketch definitions must be saved to a file to generate completed sketches.
[6]A depth of 0 produces a list containing 1 gadget.

Table 2: Synthesized gadget sequences

Depth	Order	Synthesized Sequence
0–1	AB	*none/unsatisfiable*
2	AB	G5, G3, G3
3	A	G5, G3, G8, G3
	B	G6, G2, G7, G1
4	A	G5, G3, G3, G8, G8
	B	G5, G3, G8, G6, G3
5	AB	G6, G2, G4, G2, G2, G7
6	A	G5, G8, G3, G2, G2, G3, G8
	B	G5, G8, G7, G6, G8, G3, G3

4.3 Experimental Results

With the code shown, our program generates the sequence[7] G5, G3, G3, which assigns 6 to r_1 and then subtracts 1 twice. This is indeed the shortest solution given the available gadgets. But what if we want a diversity of (possibly suboptimal) solutions? Can the synthesis algorithm discover them for us?

One approach is to increase the requested depth of the generated sequence. After changing the depth from 2 to 3, Rosette instead discovers the sequence G5, G3, G8, G3, which assigns 6 to r_1, subtracts 1, copies r_1 to r_2 (spuriously), and finally subtracts 1 again. This result goes to the heart of our observation about the nature of diversity in synthesis. We see that the new sequence is identical to our first answer, but with an additional copy gadget that acts as a semantic no-operation. The synthesis engine has thus discovered *code insertion* as an approach to the diversity problem.

Continuing the experiment by increasing the depth reveals more solutions of similar spirit. For example, Rosette's depth-4 solution is G5, G3, G3, G8, G8, which uses the semantic no-operation twice, this time both at the end. But what if we desire even greater diversity? Is there some way to encourage the search toward even more exotic regions of the search space without modifying the internals of the algorithm?

One potential way to do so is by randomly permuting the order of the gadget list offered to the search engine. This capitalizes on the fact that search algorithms tend to consider the available choices in order, in the absence of any compelling reason to do otherwise. To test this, we changed the synthax definition slightly to put G6 before G5:

```
(define-synthax (gadget-sequence depth)
  #:base (list [choose g1 g2 g3 g4 g6 g5 g7 g8])
  #:else (cons [choose g1 g2 g3 g4 g6 g5 g7 g8]
          (gadget-sequence (- depth 1))))
```

The new experimental results are reported in Table 2. In the "Order" column, the value "A" indicates a result discovered using our original gadget order, "B" indicates one discovered using the permuted order, and "AB" indicates the same answer was arrived at by both versions.

Our first new answer—at depth 3, consisting of G6, G2, G7, G1—assigns 1 to r_2, adds 1, copies it to r_1, and multiplies r_1 by 2 to arrive at the answer 4. This shows that we can get significantly different answers with only a minor modification in the synthesis. The change in gadget order has changed the "shape" of the search space—the solver goes down paths generated by two different grammars which happen to generate the same language. We conjecture that similarly low-effort adaptations of program synthesis tools could open an important new application area for this significant line of research, and offer far richer forms of artificial diversity than are presently achieved with *ad hoc* code obfuscation strategies.

Our small experiment can be seen as applicable to both offensive and defensive security. Used for offense, we can imagine this type of synthesis being used to generate equivalent but different malicious programs, which could reliably evade signature-based intrusion detection. While the limits of signature-based antivirus are already well-known (see Section 1), it is still regarded as useful for inoculating computers against the "background radiation" of the internet [63]—prevelent yet unsophisticated attacks that threaten most users on a daily basis. But if unsophisticated attacks can be effortlessly bred into highly diverse communities through program synthesis, even the unsophisticated attacks become highly threatening. In addition, prior work has observed that technologies that automate significant diversity can imbue malware with reactive capabilities—allowing software attacks to direct their mutations in order to penetrate specific defenses [25, 54].

Conversely, if used as a defensive measure, this sort of code synthesis could be a way of automatically introducing artificial macro-diversity into a software population. Synthesis research has historically sought improved program development from scratch, and our experiments illustrate how it could also be applied to diversify already-developed programs in meaningful ways. Both modalities seem promising, in that they offer to leverage decades of powerful scientific advances in programming language theory toward improving the state-of-the-art in this notoriously difficult cyber security problem space.

5. WORKSHOP DISCUSSIONS

We've tried to capture in this section key points discussed in the workshop, and clarify or address questions and topics brought up by NSPW participants.

5.1 Bugs and Security Flaws

5.1.1 Specification Problems

As mentioned in Section 1, both bugs and security flaws that are captured by the specification itself rather than the implementation will necessarily be present in all variants generated from that specification.

For example, when using sketching for code synthesis, any code outside of a hole in a sketch is reproduced exactly; errors in this part of the sketch will be retained in the final product. These parts of a sketch may be considered to be part of the specification (i.e., the parts of the program that are specified exactly). In particular, since control flow structure is likely to be outside of the holes in a sketch (as mentioned in Section 3.2), any control-flow vulnerabilities will necessarily be retained.

However, the fact that specifications may encode problems is not unique to sketching; specifications are always by defini-

[7]Specifically, it synthesizes the code fragment `(cons g5 (cons g3 (list g3)))`.

tion trusted, and any buggy formal specification will produce buggy results. Further, writing correct formal mathematical specifications may be seen in the development community as work requiring more highly trained professionals, and therefore as too costly an endeavor to undertake. The hope in the software synthesis community is that higher level specifications will be not only be clearer than traditional source code, but that the guarantee that any generated programs adhere to the specification provides its own reward.

5.1.2 Hole Complexity

When sketching, since holes are usually described by grammars, the possible contents of a filled in hole will always be exactly equal to the language generated by the grammar in question (as restricted by the assertions given). Thus, controlling the "size" or complexity of a hole is exactly equivalent to adjusting the grammar. The shape of this generated language exactly controls the search space and therefore the possible results produced in each variant.

5.1.3 Logistics

During a normal program development lifecycle, bugs (and to a lesser extent security issues) are found often and must be dealt with by the developers. Having different variants distributed to different end-users may complicate the process of finding and fixing errors.

Once a problem is discovered, the first task will be to determine whether the problem exists in all generated variants (and is therefore an error in the specification), or whether only one or a few variants exhibits the issue. In the former case, repairing the specification will fix the problem. Otherwise, several courses of action may be taken. The developer might:

- provide a patch specific to the erroneous variant(s);
- adjust the specification to prevent the error from ever being generated;
- do nothing; in some sense, statistically limiting security errors to a few variants is the goal of this work—having a limited number of deployments that could be exploited might be tolerable in some cases;
- blacklist the variant(s) in question: a mechanism or constraint could be added to the synthesis engine or specification to prevent emission of particular versions. To assist with this a token uniquely determining the variant in question could be provided at generation time (e.g., numbers representing which grammar choices were made in each hole).

The tokens mentioned in the last point above would be useful not only in preventing the generation of particular variants during synthesis, but also in recreating them for debugging and testing purposes. However, if the synthesis algorithm and specification are both known to an attacker, care may have to be taken not to expose token values as well lest the attacker be able to recreate the version in use by a particular target.

A further logistical consideration is that of code signing: if signed code is desired, each variant would have to be signed by the code producer.

5.2 Stone Soup

Workshop participants brought to our attention a large body of work related to artificial diversity: the STONESOUP [28] project. The project concerns securely executing potentially untrusted programs; artificial diversity is a common technique used to achieve this goal. Example projects in this program include: MINESTRONE [36, 59], PEASOUP [10], and VIBRANCE [7, 11, 53].

5.3 Hardware Diversification

It was mentioned in the workshop that diversification may be desirable not just at the software level, but at the hardware level as well. This is certainly possible; Rosette has been applied to synthesize circuit designs [75] in a hardware specification language, which could be diversified using the same techniques suggested in this paper.

5.4 Metrics

A metric for determining the degree of difference between two generated programs would be useful to have, both for the synthesis algorithm (which might desire a minimum distance between subsequent outputs), and for program analysis. Such a metric would be hard to develop for the same reasons that program equivalence is a hard problem.

It should be noted that any metric for diversity would not be the same as a metric for security. Finding useful metrics for security has proved to be a difficult problem as well; however, future work might determine which security properties (if any) can be inferred from the degree of difference between two program variants.

6. FUTURE WORK

Our preliminary exploration of synthesis for diversity raises many open research questions, which should be investigated by future work. We discuss several of these below, suggesting potential avenues of investigation.

Security properties of different synthesis techniques.
Synthesis algorithms are routinely evaluated in terms of accuracy and performance, but it would be valuable to additionally evaluate them from a security perspective (both defensively and offensively). For example, some techniques may generate a larger number of more diverse programs when broadened to admit multiple solutions. Defensive evaluations should consider whether this variety educes resilience against attacks typically immune to artificial micro-diversity, while offensive evaluations should measure its impact on antivirus detection rates. Trade-offs between diversity and runtime performance may also be an important consideration.

Meaningful evaluation must also take care to distinguish security from obscurity. Even if existing attacks and defenses fail against these more synthetically diverse software communities, we must carefully consider whether an attacker or defender with knowledge of the synthesis algorithm might be able to comprehensively adapt their methods to accommodate the diversity. For example, if synthesis for malware obfuscation yields populations with uniquely distinguishable structure, then small changes to malware signature databases will serve to easily mitigate the threat even if existing databases are inadequate. Similarly, if defensively synthesized software populations harbor new, yet predictable

vulnerabilities, then we've merely replaced one monoculture with another.

Synthesis as an Arms Race.

Can synthesis technologies be turned backward to reverse any ostensible security benefits they offer? For example, could defenders observing a synthetically diverse malware population reliably infer the synthesis algorithm and its parameters, and use this information to reliably detect all members of the population? Dually, could attackers apply the same synthesis techniques used to harden a software population to instead craft attacks that succeed against it with high probability? We consider this an important question for understanding synthesis' role in the software security arms race. Answers may also lend insight into the following related problems:

1. *Understanding malware intent:* The capacity to derive general requirements and capabilities specifications from a subset of known variants of a given malware family has long been a holy grail of antivirus defense. Expediting such analyses through greater automation would grant defenders a significant advantage in the malware arms race.

2. *Clustering similar malware:* The requirements specifications of different malware samples can be compared as an additional signal when determining malware similarity for the purposes of classification into families.

3. *Generating new malware from existing samples:* An offensive use of reversible synthesis would be to generate requirements specifications from known malware families and to use them to generate diversified variants. This would be a powerful tool to a motivated attacker, as they could reuse the vast archives of malware that already exist to create newer, stealthier versions.

Specification-free Software.

Most widely deployed (and widely attacked) software products have no formal specification. Despite copious software engineering research underscoring its importance, formal specification remains an embarrassing omission in most software development lifecycles—even in products that find their way into mission-critical systems. Can program synthesis nevertheless diversify these specification-free products?

There is good reason to expect the answer might be affirmative. For example, program sketches—exhibited by our experiments in Section 4—are forms of specification that are relatively easy to derive from reference implementations (e.g., by replacing portions of the reference implementation with holes). Future research should therefore investigate the feasibility of leveraging these approaches to diversify legacy software.

7. CONCLUSION

In this paper, we make the case for research that combines two expansive but as of yet separate fields of study—artificial diversity and automated program synthesis.

Artificial diversity allows different versions of a software to have identical semantics but different syntactic representations, and in doing so offers a powerful tool that is used for both offensive and defensive security.

Defensively it introduces uncertainty into a target binary that provides probabilistic guarantees against multiple classes of attacks. Offensively it provides a way for malware to evade detection.

Automated program synthesis attempts to systematically derive a program from a higher level specification. Many synthesis algorithms also allow formal assertions to be made about the correctness of the derived program with respect to its specification. However, existing literature on synthesis has not yet explored creating multiple variants from a single specification.

We use the Rosette language to perform some preliminary experiments towards the applicability of program synthesis to introduce diversity into a program population. The results show that there is indeed promise in this approach, and motivates the need for further research in this area.

There are multiple open questions that need to be answered before the utility of this approach can be accurately quantified, but we believe this also has the potential to improve both the offensive and defensive state of the art.

8. ACKNOWLEDGMENTS

The research reported herein was supported in part by NSF awards #1054629 and #1513704, ONR award N00014-14-1-0030, and an NSF I/UCRC award from Raytheon Company. Any opinions, recommendations, or conclusions expressed are those of the authors and not necessarily of NSF, ONR, or Raytheon. Thanks go to Sean Peisert for shepherding our paper for the workshop and for his help throughout the process; to our discussion chair Anil Somayaji; to the anonymous reviewers for their time spent and helpful comments; and finally to all the participants at NSPW for their lively discussion of the topic.

9. REFERENCES

[1] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proc. 8th IEEE Int. Fault Tolerant Computing Sym. (FTCS)*, pages 3–9, 1978.

[2] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn. FlashRelate: Extracting relational data from semi-structured spreadsheets using examples. In *Proc. 36th ACM Conf. Programming Language Design and Implementation (PLDI)*, pages 218–228, 2015.

[3] B. Baudry and M. Monperrus. The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Computing Surveys (CSUR)*, 48(1), 2015.

[4] A. W. Biermann, R. I. Baum, and F. E. Petry. Speeding up the synthesis of programs from traces. *IEEE Trans. Computers*, 100(2):122–136, 1975.

[5] D. Bodeau and R. Graubart. Cyber resiliency engineering framework. Technical Report MTR110237, MITRE Corporation, 2011.

[6] D. Bodeau and R. Graubart. Cyber resiliency assessment: Enabling architectural improvement. Technical Report MTR120407, MITRE Corporation, 2013.

[7] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Verified integrity properties for safe approximate program transformations. In *Proc. ACM SIGPLAN*

Work. on Partial evaluation and program manipulation, pages 63–66, 2013.

[8] K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing formal specifications using testing. In *Proc. 4th Int. Conf. Tests and Proofs (TAP)*, pages 6–21, 2010.

[9] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[10] M. Co, J. W. Davidson, J. D. Hiser, J. C. Knight, A. Nguyen-Tuong, D. Cok, D. Gopan, D. Melski, W. Lee, C. Song, T. Bracewell, D. Hyde, and B. Mastropietro. PEASOUP: Preventing exploits against software of uncertain provenance (position paper). In *Proc. 7th ACM Int. Work. Software Engineering for Secure Systems (SESS)*, pages 43–49, 2011.

[11] A. Coglio, M. Becker, S. Fitzpatrick, L. Gilham, C. Green, E. McCarthy, H. Sipma, M. Barry, A. Browne, E. Bush, D. Smith, A. Venet, M. Rinard, J. Perkins, J. Eikenberry, D. Kramm, P. Piselli, D. Willenson, S. Misailovic, F. Long, M. Carbin, R. Laddaga, P. Robertson, and P. Manghwani. Vulnerabilities in bytecode removed by analysis, nuanced confinement and diversification (VIBRANCE). Technical Report AFRL-RY-WP-TR-2015-0019, Defense Technical Information Center (DTIC), 2015.

[12] F. Cohen. *Computer Viruses*. PhD thesis, University of Southern California, 1986.

[13] F. B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, 1993.

[14] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. 25th ACM Sym. Principles Of Programming Languages (POPL)*, pages 184–196, 1998.

[15] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Proc. 15th USENIX Security Sym.*, pages 105–120, 2006.

[16] A. Cypher and D. C. Halbert. *Watch What I Do: Programming By Demonstration*. MIT press, 1993.

[17] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *Proc. 8th ACM SIGSAC Sym. Information, Computer and Communications Security (ASIACCS)*, pages 299–310, 2013.

[18] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. https://racket-lang.org/tr1.

[19] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proc. 6th Work. Hot Topics in Operating Systems (HotOS)*, pages 67–72, 1997.

[20] M. Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proc. New Security Paradigms Work. (NSPW)*, pages 7–16, 2010.

[21] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Proc. 18th Int. World Wide Web Conf. (WWW)*, pages 561–570, 2009.

[22] S. Gulwani. Dimensions in program synthesis. In *Proc. 12th Int. ACM SIGPLAN Sym. Principles and Practice of Declarative Programming (PPDP)*, pages 13–24, 2010.

[23] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Component based synthesis applied to bitvector circuits. Technical Report MSR-TR-2010-12, Microsoft Research, 2010.

[24] S. Gulwani and M. Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proc. ACM SIGMOD Int. Conf. Management Data*, pages 803–814, 2014.

[25] K. W. Hamlen. Stealthy software: Next-generation cyber-attacks and defenses, invited paper. In *Proc. 11th IEEE Intelligence and Security Informatics Conf. (ISI)*, pages 109–112, 2013.

[26] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *Proc. 33rd IEEE Sym. Security & Privacy (S&P)*, pages 571–585, 2012.

[27] J. P. Inala, X. Qiu, B. Lerner, and A. Solar-Lezama. Type assisted synthesis of recursive transformers on algebraic data types. *CoRR*, abs/1507.05527, 2015.

[28] Intelligence Advanced Research Projects Activity (IARPA). Securely taking on new executable software of uncertain provenance (STONESOUP). https://www.iarpa.gov/index.php/research-programs/stonesoup. Accessed: Oct. 2016.

[29] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *Proc. 10th Int. Conf. Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 36–46, 2010.

[30] A. R. Ives and S. R. Carpenter. Stability and diversity of ecosystems. *Science*, 317(5834):58–62, 2007.

[31] J. Jeon, X. Qiu, J. S. Foster, and A. Solar-Lezama. JSketch: Sketching for Java. In *Proc. 10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 934–937, 2015.

[32] J. Jeon, X. Qiu, A. Solar-Lezama, and J. S. Foster. Adaptive concretization for parallel program synthesis. In *Proc. 27th Int. Conf. Computer Aided Verification (CAV)*, pages 377–394, 2015.

[33] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proc. 32nd ACM/IEEE Int. Conf. Software Engineering (ICSE)*, volume 1, pages 215–224, 2010.

[34] S. Katayama. Systematic search for lambda expressions. In M. van Eekelen, editor, *Trends in Functional Programming*, volume 6, pages 111–126. Intellect Books, University of Chicago Press, 2005.

[35] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. 10th ACM Conf. Computer and Communications Security (CCS)*, pages 272–280, 2003.

[36] A. D. Keromytis, S. J. Stolfo, J. Yang, A. Stavrou, A. Ghosh, D. Engler, M. Dacier, M. Elder, and D. Kienzle. The MINESTRONE architecture combining static and dynamic analysis techniques for software security. In *Proc. 1st IEEE SysSec Work. (SysSec)*, pages 53–56, 2011.

[37] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In

Proc. 13th USENIX Security Sym., 2004.

[38] C. Kreibich and J. Crowcroft. Honeycomb: Creating intrusion detection signatures using honeypots. *ACM SIGCOMM Computer Communication Review*, 34(1):51–56, 2004.

[39] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *Proc. 35th IEEE Sym. Security & Privacy (S&P)*, pages 276–291, 2014.

[40] T. Lau, P. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In *Proc. 2nd Int. Conf. Knowledge Capture (K-CAP)*, pages 36–43, 2003.

[41] V. Le and S. Gulwani. FlashExtract: A framework for data extraction by examples. In *Proc. 35th ACM Conf. Programming Language Design and Implementation (PLDI)*, pages 542–553, 2014.

[42] V. Le, S. Gulwani, and Z. Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proc. 11th Annual Int. Conf. Mobile Systems, Applications, and Services (MoviSys)*, pages 193–206, 2013.

[43] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proc. 27th IEEE Sym. Security & Privacy (S&P)*, pages 15–47, 2006.

[44] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th ACM Conf. Computer and Communications Security (CCS)*, pages 290–299, 2003.

[45] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–45, 2007.

[46] R. MacArthur. Fluctuations of animal populations and a measure of community stability. *Ecology*, 36(3):533–536, 1955.

[47] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Communications of the ACM (CACM)*, 14(3):151–165, 1971.

[48] Z. Manna and R. J. Waldinger. A deductive approach to program synthesis. *ACM Trans. Programming Languages And Systems (TOPLAS)*, 2(1):90–121, 1980.

[49] H. Massalin. Superoptimizer: A look at the smallest program. In *Proc. 2nd Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–126, 1987.

[50] R. M. May. *Stability and Complexity in Model Ecosystems*, volume 6. Princeton University Press, 1973.

[51] S. McLaughlin, D. Podkuiko, A. Delozier, S. Miadzvezhanka, and P. McDaniel. Embedded firmware diversity for smart electric meters. In *Proc. 5th USENIX Conf. Hot Topics in Security (HotSec)*, 2010.

[52] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai. A machine learning framework for programming by example. In *Proc. 30th Int. Conf. Machine Learning (ICML)*, pages 187–195, 2013.

[53] S. Misailovic and M. Rinard. Synthesis of randomized accuracy-aware map-fold programs. Technical Report MIT-CSAIL-TR-2013-031, MIT Computer Science and Artificial Intelligence Laboratory (CSAIL), 2013.

[54] V. Mohan and K. W. Hamlen. Frankenstein: Stitching malware from benign binaries. In *Proc. 6th USENIX Work. Offensive Technologies (WOOT)*, pages 77–84, 2012.

[55] L. D. Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. 14th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[56] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proc. 26th IEEE Sym. Security & Privacy (S&P)*, pages 226–241, 2005.

[57] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 772–781, 2013.

[58] P. O'Kane, S. Sezer, and K. McLaughlin. Obfuscation: The hidden malware. *IEEE Security & Privacy*, 9(5):41–47, 2011.

[59] V. Pappas, M. Polychronakis, and A. D. Keromytis. Practical software diversification using in-place code randomization. In S. Jajodia, A. K. Ghosh, V. S. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, editors, *Moving Target Defense II*, volume 100 of *Advances in Information Security*, pages 175–202. Springer, 2013.

[60] S. Peisert, E. Talbot, and M. Bishop. Turtles all the way down: A clean-slate, ground-up, first-principles approach to secure systems. In *Proc. New Security Paradigms Work. (NSPW)*, pages 15–26, 2012.

[61] O. Polozov and S. Gulwani. LaSEWeb: Automating search strategies over semi-structured web data. In *Proc. 20th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, pages 741–750, 2014.

[62] B. Randell. System structure for software fault tolerance. In *Proc. Int. Conf. Reliable Software*, pages 437–449, 1975.

[63] B. Schneier. Is antivirus dead? *Schneier on Security*, May 2014.

[64] E. M. Schulte, W. Weimer, and S. Forrest. Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair. In *Proc. Companion Publication Annual Conf. Genetic and Evolutionary Computation (GECCO)*, pages 847–854, 2015.

[65] E. J. Schwartz and D. Brumley. Q: Exploit hardening made easy. In *Proc. 20th USENIX Security Sym.*, 2011.

[66] E. Shioji, Y. Kawakoya, M. Iwamura, and T. Hariu. Code shredding: Byte-granular randomization of program layout for detecting code-reuse attacks. In *Proc. 28th Annual Computer Security Applications Conf. (ACSAC)*, pages 309–318, 2012.

[67] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proc. 6th USENIX Sym. Operating Systems Design and Implementation (OSDI)*, 2004.

[68] A. Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, The University of California, Berkeley, 2008.

[69] A. Solar-Lezama. The sketching approach to program synthesis. In *Proc. 7th Asian Sym. Programming Languages and Systems (APLAS)*, pages 4–13, 2009.

[70] A. Somayaji, M. Locasto, and J. Feyereisl. The future of biologically-inspired security: Is there anything left

to learn? In *Proc. New Security Paradigms Work. (NSPW)*, pages 49–54, 2008.

[71] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. Foster. Program inversion revisited. Technical Report MSR-TR-2010-34, Microsoft Research, 2010.

[72] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Proc. 37th ACM SIGPLAN-SIGACT Sym. Principles of Programming Languages (POPL)*, pages 313–326, 2010.

[73] C. Taylor and J. Alves-Foss. Diversity as a computer defense mechanism. In *Proc. New Security Paradigms Work. (NSPW)*, pages 11–14, 2005.

[74] E. Torlak. The Rosette language. https://emina.github.io/rosette. Accessed: Apr. 2016.

[75] E. Torlak and R. Bodik. Growing solver-aided languages with Rosette. In *Proc. ACM Int. Sym. New Ideas, New Paradigms, and Reflections Programming & Software (Onward!)*, pages 135–152, 2013.

[76] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proc. 35th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pages 530–541, 2014.

[77] K. Wang, J. J. Parekh, and S. J. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Proc. 9th Int. Sym. Recent Advances in Intrusion Detection (RAID)*, pages 226–248, 2006.

[78] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. In *Proc. 7th Int. Sym. Recent Advances in Intrusion Detection (RAID)*, pages 203–222, 2004.

[79] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. 19th ACM Conf. Computer and Communications Security (CCS)*, pages 157–168, 2012.

[80] D. Yadron. Symantec develops new attack on cyberhacking: Declaring antivirus software dead, firm turns to minimizing damage from breaches. *The Wall Street Journal*, May 2014.