Send Orders for Reprints to reprints@benthamscience.ae

BENTHAM OPEN

CrossMark

The Open Artificial Intelligence Journal

Content list available at: www.benthamopen.com/TOAIJ/

DOI: 10.2174/1874061801805010001, 2018, *5*, 1-13

RESEARCH ARTICLE

Knowledge-Based Programming for the Cybersecurity Solution

Stuart H. Rubin*

Space and Naval Warfare Systems Center Pacific, San Diego, CA 92152-5001, USA

Received: August 7, 2018 Revised: October 22, 2018 Accepted: November 25, 2018

Abstract:

Introduction:

The problem of cyberattacks reduces to the unwanted infiltration of software through latent vulnerable access points. There are several approaches to protection here. First, unknown or improper system states can be detected through their characterization (using neural nets and/or symbolic codes), then interrupting the execution to run benchmarks and observe if they produce the states they should. If not, the execution can be rewound to the last successful benchmark, all states restored, and rerun.

Methods:

This will only work for cyber-physical systems that can be rewound. Benchmarks will often include sensory information. The second approach is termed, "semantic randomization". This is similar to the well-known compiler technique known as "syntactic randomization". The significant difference is that different variants of the algorithm itself are being automatically programmed. Cyberattacks will generally not be successful at more than one variant. This means that cybersecurity is moving us towards automatic programming as a desirable consequence. Knowledge-Based Software Engineering (KBSE) is the way to achieve automatic programming in practice.

Discussion:

There is non-determinism in the execution of such systems, which provides cybersecurity. Knowledge-based algorithmic compilers are the ultimate solution for scalable cost-effective cybersecurity. However, unlike the case for the less-secure syntactic randomization, the cost-effectiveness of semantic randomization is a function of scale. A simple randomization-based automatic programming method is illustrated and discussed.

Conclusion:

Semantic randomization is overviewed and compared against other technologies used to protect against cyberattack. Not only does semantic randomization itself, or in combination with other methodologies, offer improved protection; but, it serves as the basis for a methodology for automatic programming, which in turn makes the semantic randomization methodology for cybersecurity cost-effective.

Keywords: Automatic programming, Code benchmarks, Grammatical inference, Knowledge-Based Software Engineering (KBSE), Search control knowledge, Semantic randomization, Syntactic randomization.

1. INTRODUCTION

Software, which is complex enough to be capable of self-reference, cannot be proven valid [1]. It similarly follows that complex software is inherently subject to cyberattack. It is just a matter of degree.

The classic approach to detecting and preventing cyberattack is to interrupt executing software at random points and run benchmark programs, whose (intermediary) output is known. If any such runs deviate from the expected output, it

* Address correspondence to this author at the Space and Naval Warfare Systems Center Pacific, San Diego, CA 92152 - 5001, USA; Tel: (619)553-3554; E-mail: stuart.rubin@navy.mil

follows that the system is infected. In such cases, one needs to rewind to the last successful benchmark, restore the context, and execute from there on. The idea is to purge the infection and continue processing from the last successful point, while continuing the random benchmark tests. Software may be kept "clean" by storing it in ROM and taking care to insure that the ROM cannot be reprogrammed. The use of benchmarks follows from "syntactic randomization" because, in general, code compiled on different compilers will not be all susceptible to the same cyberattack, which is then detectable and recoverable from [2]. This approach works well for cyber-physical systems that can be rewound. A car at a spoofed stop sign is not representative of such a system. Benchmarks need to include sensory information, which is typically included in the rewind. There is an opportunity to insert machine learning here because false sensor readings can be learned and generalized if symbolic learning is used. More-complex cyberattacks require that the sequence of system states be used as a context for the detection of a cyberattack.

Next, while syntactic randomization offers cybersecurity in proportion to the number of distinct compilers used, the security of this approach can be amplified by modifying the syntax of the source code (*i.e.*, not the semantics) so that a different version of the same algorithm, having the same computational complexity, computes the same benchmarks. Not only does this provide more cybersecurity than syntactic randomization, but such "semantic randomization" is compatible with the concomitant use of syntactic randomization for even greater security. Moreover, the greater the number of non-deterministic benchmarks, the greater the provided security [3, 4].

Semantic variants can be costly to provide. That is why semantic randomization is not more widely used as a way to increase cybersecurity. However, they can be supplied by automatic programming as a phenomenon of scale. The larger the scale, the less the cost per Source Line of Code (SLOC). Semantic randomization can be achieved by introducing random non-determinism in the synthesis of code. Not every line of code need be automatically programmed. Many are simply reused from a set of class-equivalent functions.

This brings us to our goals for this paper; namely, as follows. Semantic randomization can be used to increase the security of code against cyberattack. This entails the creation of code "copies" (*i.e.*, modules, functions, methods, and/or procedures), which compute the same semantics as the originals in comparable space and time, while being syntactically distinct. This requirement for syntactic distinction, follows from randomization theory, and informally means the equivalent of having been written by different people, vice interchanging x and y, or the like, throughout. Simply put, the equivalent semantics must be computed by a novel syntax. Of course, this cybersecurity could be achieved by a team of redundant programmers; but, that obviously would not be a cost-effective approach. The aim of this paper is to introduce a schema-based technique for such automatic programming, which makes practical a cost-effective approach to cybersecurity. It is worth noting that the goal of automatic programming is not the total displacement of the human in the loop by the machine; rather, the implication is that the human should do what he or she does best and the machine likewise. This means that people should focus on cognitive tasks, while the machine does repetitive or search-based tasks. The programmer will specify a (tractably searchable) space of I/O constrained programs (*i.e.*, schemas); and, the machine will search out one or more program instances of that space in satisfaction of all I/O constraints. In this manner, automatic "symmetric" programming is not only possible, but cost-effective as well. This makes semantic randomization based cybersecurity cost-effective. By way of contrast, syntactic randomization, a well-known cybersecurity technique, is inexpensive. It merely entails the use of distinct compilers to produce distinct object codes, but is not as effective as the semantic randomization technique, which better-thwarts cyberattacks through the insertion of knowledge expressed as alternative code syntax. This paper offers one way to minimize the cost of so doing. Multiple cybersecurity techniques can be combined for even greater cost-effective cybersecurity. This paper also shows the cybersecurity problem in a new light. That is, nature has allowed the cybersecurity problem to exist in order to further the development of automatic programming techniques, which will be addressed below.

2. MATERIALS AND METHODS

2.1. The Automatic Programming Solution

Knowledge-Based Software Engineering (KBSE) is a technique, which applies expert systems to the compilation of code. These systems are also known by the term, "expert compiler" [5].

Automatic programming is not new technology [6, 7]. What is relatively new is the scalability of the same through the use of multiple cores and distributed server computation.

Automatic programming requires the use of some sort of specification language. This can serve as a port of entry for cyber intrusion. What is needed here are multiple specification languages, or at least compilations, to defray the

effectiveness of cyberattacks. The generation of KBSE is a spiral development process. Thus, cyber protection requires domain understanding. One solution is distributed software synthesis. Here, one expert compiler calls another in a graph representation of invocation patterns. Notice that algorithmic synthesis depends upon reuse at the highest or most-abstract level. This effectively serves as a highest-level programming language with the side-benefit of providing relatively cyber-secure output codes as previously described.

It follows that the science and particularly the engineering of cyber-secure systems is a consequence of building expert compilers. Such compilers will need to create functions (*i.e.*, functional programming) to insure interchangeability among the various functions. Global variables, other than those connected with blackboards and non-monotonic reasoning, are notoriously difficult to compile. That is the reason why the scope of synthesized code and variables need be local as a practical matter.

One way to write code, which writes code, is to build a distributed networked expert compiler, which synthesizes some space of functional programs. The code to be synthesized first are rules for synthesizing code. Here, design decisions can be generalized to provide more options for the synthesized rules. This is a form of bootstrapping for cracking the knowledge acquisition bottleneck [7].

The way to write meta-rules is to group rules into segments, which are applied by other rules [8]. Segments are cohesive; and, as a consequence, rules and groups of rules can be grouped by chance on the basis of synthesizing code, which functions in satisfaction of sets of I/O constraints. While the validation of arbitrary code can be achieved by expert systems of arbitrary complexity, functional code is amenable to I/O testing by making use of orthogonal sets of I/O pairs. A pair of I/O pairs is said to be orthogonal with respect to a software function if the two I/O tests do not share the same execution path trace. In practice, such pairings of I/O tests need only be relatively orthogonal as a cost-saving measure. For example, an orthogonal set of sort program tests is {((3 2 1) (1 2 3)) ((3 1 2) (1 2 3)) ((1 2 3) (1 2 3))}. A pair of non-orthogonal tests is, for example, {((2 1) (1 2)) ((3 2 1) (1 2 3))}. The key to identifying relatively orthogonal test vectors is that the set of them is random in the recursively enumerable sense [9]. That is, the test vectors should not be functionally related to each other. Validated functions can be further composed and similarly tested for novel functionality.

So far, we have seen that the problem of providing practical cybersecurity is tied to the problem of automatic programming. This problem, in turn, is tied to the problem of knowledge acquisition.

Knowledge acquisition goes beyond neural networks. Indeed, effective knowledge acquisition by humans involves transformational analogy, which is something that neural networks cannot do [10, 11]. This then is the domain for symbolic AI, since neural networks inherently cannot perform basic “modus ponens”.

The key to successful KBSE is having as much knowledge in as many representational formalisms [12] as possible available to bear. In theory, one needs as high a density of knowledge as possible, which cannot be bounded [13].

3. ON RANDOMIZATION

The density of knowledge can be increased, in theory, by applying knowledge to itself, as previously indicated. One can generalize data and generalize generalizations. When this leads to relatively incompressible levels, the applicable term is “randomization” [9]. For example, numerical sequences are said to be “random” because no more concise generating algorithm than the sequence itself is known.

Random knowledge has the property that the converse (*i.e.*, symmetric knowledge) follows from the application of one random knowledge function to another - including itself. For example, the add-one function increases its numerical input by one. The mod-two function returns true just in case its numerical input is even. The composition of the mod-two and add-one function returns true just in case its numerical input is odd. This is a symmetric function, which is defined by a composition of two relatively random (*i.e.*, orthogonal) functions. Notice that this is the way to expand the knowledge base. This expansion, in turn, increases the space of functions, which can be automatically programmed. This space includes many non-deterministic variants too, which provide benchmarked security *via* semantic randomization. Relatively random functions are recursively enumerable because there is no known composition, which can express them [9].

The next question might be, “which is to be preferred - an expert compiler, or a case-based compiler for KBSE?” The answer is that it follows from randomization theory that both are required [13]. Cases serve as a repository for experiential knowledge. They can be generalized into rules through the application of cases and/or rules once their

domain is understood. This is not necessarily achievable in practice. However, the chance of a successful generalization is maximized through the use of networked coherent systems. Coherent systems are relatively random. The point is that such systems of systems need to be engineered not only to advance the goals of KBSE, but those of cybersecurity as well. Knowledge spawning knowledge may sound simple enough, but it represents perhaps the most complex project ever conceived.

It is well-known that the representation of knowledge and data determines that, which can be learned [12]. Neural networks can map inputs to outputs, but they can neither deduce nor induce new knowledge. That requires symbolic approaches; including deduction, induction, abduction, and fuzzy pattern matching. In particular, symbolic approaches are compatible with grammatical inference [14, 15], which is tied to programming language definition. All these methods and more are related through randomization [9, 13]. Indeed, it is postulated that deficiencies in today's neural networks [10, 11] will be ameliorated through the redefinition of neural networks using randomization theory [13].

4. EXAMPLE OF CYBERSECURE SOFTWARE SYNTHESIS

The problem is to detect a cyberattack when it happens and recover from a cyberattack while it happens. Software needs to be subdivided into components, which map a set of input vectors to a non-deterministic set of stochastic output vectors. Components are defined in terms of other components, which are defined by rules (Fig. **1**).

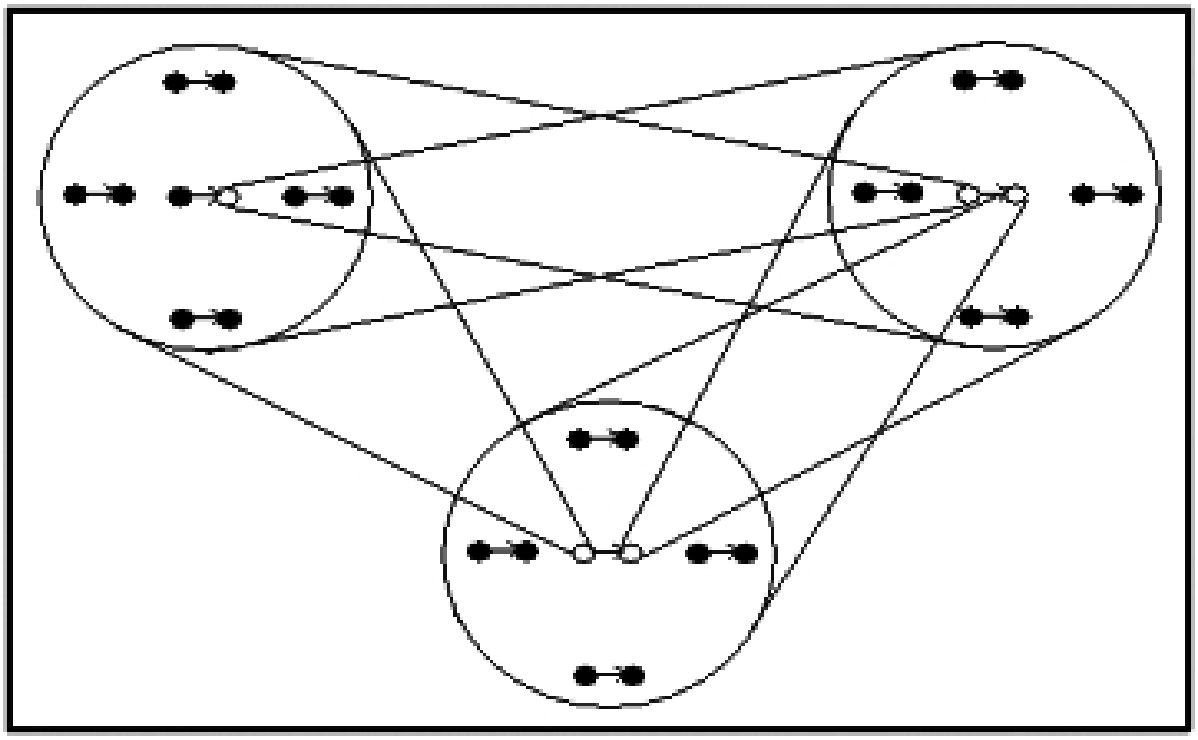

Fig. (1). Recursive rule-based definition of software components.

The behavior of a set of Boolean components or a sequence of procedural components is not unique. Thus, it is possible to synthesize a diverse set of components, which provides the desired security for an arbitrary I/O characterization.

4.1. Justification for I/O Characterization of Software Components

It is acknowledged that there is a software, which cannot be sufficiently characterized by a non-deterministic stochastic I/O mapping. For example, a component might draw a picture. Here, a knowledge-based and/or neural system may be applied to rank the quality of the component. In a sense, mapping input to desired output(s) is universal, it's just that intermediate evaluation code is sometimes needed. Thus, while we will not address such complexities in this paper, it is to be understood that the methodology advanced herein is completely compatible with them. In fact, it may be used to define the intermediate knowledge-based evaluation systems.

Another point of contention pertains to the use of empirical testing instead of, or in combination with, denotational or axiomatic semantics for program validation. The recursive Unsolvability of the Equivalence Problem [1] proves that in the general case it is impossible to prove that two arbitrary programs compute the same function. Moreover, approaches to program validation, based on computational semantics, have proven to be unacceptably difficult to apply in practice. There can be no theoretical method for insuring absolute validity once a program grows to a level of complexity to be capable of self-reference [1, 13, 16, 17].

It follows that program validation is properly based on empirical testing, the goal of which is to cover a maximal number of execution paths using a minimal number of test cases. This is none other than randomization [9, 13]. Of course, there is no need to achieve the absolute minimum here; a minimum relative to the search time required to find the test cases will suffice. In a large enough system of systems, the methodology advanced herein may be applied to the generation of relatively random test cases. Randomization serves to maximize reuse. Reuse is perhaps the best real-world technique for exposing and thus minimizing the occurrence of program bugs.

4.2. Random-Basis Testing

Each component, saved in the database, is associated with one or more I/O test vector pairings that serve to map a random input vector to correct non deterministic output vectors. The underpinning principle is that test vectors, which have been sufficiently randomized, are relatively incompressible. For example, consider the synthesis of a sort function using LISP (Fig. **2**). There are some extraneous details, such as knowing when a particular sequence will lead to a stack overflow; but, these are easily resolved using an allowed execution time parameter. Impressive programs have been so synthesized; supporting the component-based concept. Notice that components can be written at any scale from primitive statements to complex functions. Given only so much allocated search time, the system will either discover a solution or report back with failure. This is in keeping with the recursive Unsolvability of the Halting Problem [1, 13].

```
((DEFUN MYSORT (S)
  (COND ((NULL S) NIL)
    (T (CONS (MYMIN S (CAR S)) (MYSORT (REMOVE (MYMIN S (CAR S)) S)))))))
? io
((((1 3 2)) (1 2 3)) (((3 2 1)) (1 2 3)) (((1 2 3)) (1 2 3)))
? (pprint (setq frepos '((CRISPY'
       (DEFUN MYSORT (S)
        (COND
        (FUZZY
         ((NULL S) NIL)
         ((ATOM (FUZZY S ((FUZZY CAR CDR) S))) NIL))
        (T (CONS (MYMIN S (CAR S))
        (MYSORT (REMOVE (MYMIN S (CAR S)) S)))))))))


((CRISPY '(DEFUN MYSORT (S)
   (COND (FUZZY ((NULL S) NIL) ((ATOM (FUZZY S ((FUZZY CAR CDR) S))) NIL))
    (T (CONS (MYMIN S (CAR S)) (MYSORT (REMOVE (MYMIN S (CAR S)) S)))))))

; Note that (ATOM S) was automatically programmed using the large fuzzy function
space.

? (pprint (auto frepos io))

((DEFUN MYSORT (S)
  (COND ((ATOM S) NIL)
    (T (CONS (MYMIN S (CAR S)) (MYSORT (REMOVE (MYMIN S (CAR S)) S)))))))

; Note that each run may create syntactically different, but semantically equivalent
functions:

? (pprint (auto frepos io))

((DEFUN MYSORT (S)
  (COND ((NULL S) NIL)
    (T (CONS (MYMIN S (CAR S)) (MYSORT (REMOVE (MYMIN S (CAR S)) S)))))))
```

Fig. (2). Function synthesis using random-basis testing.

Consider such I/O constraints as (((3 2 1) (1 2 3)) ((3 1 2) (1 2 3))). That is, when (3 2 1) is input to the sort function, it is required to output (1 2 3). Similarly, when (3 1 2) is input to it, it is required to output the same. Clearly, there is little value in using a test set such as (((1) (1)) ((2 1) (1 2)) ((3 2 1) (1 2 3)) ((4 3 2 1) (1 2 3 4)) …).

The purpose served by case I/O exemplar vectors, such as illustrated above, is to filter out synthesized functions, which do not satisfy the specified mapping constraints. A super set of such functions can thus be filtered down to a non-deterministic set of functions. The I/O test vectors must be mutually random so as to maximize the number of distinct execution paths tested. This helps to insure that the selected function(s) will perform, as desired, on unspecified input vectors. Moreover, in as much as every function is synthesized as a random instance of its parent schema, semantically equivalent non-deterministic functions are likely to be syntactically distinct. Almost certainly, they will be syntactically distinct in the context of other synthesized and composed functions. This is sufficient to insure cyber security.

One problem, which occurs in the synthesis of large complex functions is that the candidate instantiation space is too large to permit tractable testing. One solution to this problem follows from the inverse triangle inequality. This means that one can effect an exponential reduction in the search space for functional instances by moving the articulation points into distinct functions, which can always be done. This moves the increase in complexity for the moved articulation point from the product of possible instances for that articulation point to the sum of possible instances for that articulation point. When this is carried out for several articulation points, in the same function, the cumulative savings in the size of the search space is clearly exponential. Moreover, the search space for function synthesis can be further reduced by independently synthesizing each function in a composition of functions and then separately validating the composition. Another technique for saving time is to evaluate each salient set of I/O constraints to the point of failure, if any. Just as is the case with "or" and "and" functions, there is no need to evaluate past the point of failure. Given that the vast majority of synthesized functions will be failures, this one strategy can collectively save considerable time. Another very-important approach to delimiting search is to employ search control heuristics. For example, in Pascal, every end keyword needs to be preceded by an unmatched begin keyword. This knowledge, and knowledge like it, can be applied to delimit the members of the matching set articulation point. Search control heuristics enable the scaling of the methodology. This is a key concept. It should be noted how semantic randomization depends, in no small part, on knowledge insertion (*i.e.*, unlike syntactic randomization). As Francis Bacon once stated, "knowledge is power"; in this case, power to thwart otherwise successful cyber-attacks.

A problem is that the test set is relatively symmetric or compressible into a compact generating function. A fixed-point or random test set is required instead and the use of such relatively random test sets is called, random-basis testing [18]. While the need for functional decomposition remains, under random-basis testing, the complexity for the designer is shifted from writing code to writing search schema and relatively random tests. For example, such a test set here is (((1) (1)) ((2 1) (1 2)) ((3 1 2) (1 2 3)) ((1 2 3) (1 2 3))). Many similar ones exist. One may also want to constrain the complexity of any synthesized component (*e.g.,* Insertion Sort, Quicksort, *et al*.). This can be accomplished through the inclusion of temporal constraints on the I/O behavior (*i.e.,* relative to the executing hardware and competing software components) [19].

4.3. Component Definition

There are two categories of components; Boolean components, which return True or False and procedural components, which compute all other functions and can post and/or retract information to/from a blackboard. There are two blackboards; a local blackboard, which is only accessible to local component functions and procedures as well as those invoked by them and a global blackboard, which is accessible to all component functions and procedures. The blackboards dynamically augment the input vectors to provide further context.

All components are composed of rules, each of which consists of one or a conjunction of two or more Boolean components, which imply one or a sequence of two or more, procedural components; including global and local RETRACT and POST. Given an input vector and corresponding output vector(s), the rule base comprising the component must map the former to that latter at least tolerance percent of the time. The default tolerance is 100 percent. Transformation may also favor the fastest component on the same I/O characterization. Notice that greater diversification comes at an allowance for less optimization.

Cyberattacks are likely to affect one LISP construct (*e.g*., null S) and not another (*e.g*., nil) or *vice versa*. Given the plethora of such constructs, which comprise a typical function, some versions of syntax will remain unscathed by a cyberattack, while others will not. This is always detectable; and, processing can be rewound and continued. It follows

that automatic programming enables cybersecurity; and, cybersecurity serves to advance automatic programming. It seems as though nature has provided us with a constructive consequence supporting the plethora of cyberattacks! That is, they serve to advance technologies for automatic programming, which is applied to ameliorate the cybersecurity problem.

4.4. Component Synthesis

A library of universal primitive and macro components is supplied and evolved. There are three ways that these are retrieved. First, is by name. Second is by mapping an input vector closer, by some definition (*e.g.,* the 2-norm *et al*.), to a desired non deterministic output vector (*i.e*., hill climbing - non contracting transformations reducing the distance to a goal state with each substitution). Third is just by mapping the input vector using contracting and non-contracting transformations (*i.e*., Type 0 transformation). Hill climbing and Type 0 transformation may be combined and occur simultaneously until interrupted. The former accelerates reaching a desired output state, while the latter gets the system off of non-global hills.

Macro components are evolved by chance. They comprise a Very High Level Language (VHLL). For example, a macro component for predicting what crops to sow will no doubt invoke a macro component for predicting the weather. Similarly, a macro component for planning a vacation will likewise invoke the same macro component for predicting the weather (*i.e*., reuse) [20].

Test vectors are stored with each indexed component to facilitate the programmer in their creation and diversification as well as with the overall understanding of the components function. While increasing the number of software tests is generally important, a domain-specific goal is to generate mutually random ordered pairs [18]. Components in satisfaction of their I/O test vectors are valid by definition. Non deterministic outputs are not stochastically defined for testing as it would be difficult to know these numbers as well as inefficient to run such quantitative tests.

As software gets more complex, one might logically expect the number of components to grow with it. Actually, the exact opposite is true. Engineers are required to obtain tighter integration among components in an effort to address cost, reliability, and packaging considerations, so they are constantly working to decrease the number of software components but deliver an ever-expanding range of capabilities. Thus, macro components have great utility. Such randomizations have an attendant advantage in that their use; including that of their constituent components implies their increased testing by virtue of their falling on a greater number of execution paths [9, 13, 19]. The goal here is to cover the maximum number of execution paths using the relatively fewest I/O tests (*i.e.,* random-basis testing [18]).

The maximum number of components, in a rule, as well as the maximum number of rules in a component, is determined based on the speed, number of parallel processors for any fixed hardware capability, and the complexity of processing the I/O vectors. It is assumed that macro components will make use of parallel/distributed processors to avoid a significant slowdown. Components that are not hierarchical are quite amenable to parallel synthesis and testing.

Components may not recursively (*e.g*., in a daisy chain) invoke themselves. This is checked at definition time through the use of an acyclic stack of generated calls. Searches for component maps are ordered from primitive components to a maximal depth of composition, which is defined in the I/O library. This is performed to maximize speed of discovery. The components satisfying the supplied mapping characterizations are recursively enumerable.

Software engineers can supply external knowledge, which is captured for the specification of components. Components are defined using a generalized language based on disjunction. This is because it is easier to specify alternatives (*i.e.,* schemas) in satisfaction of I/O constraints than to specify single instances (*e.g.,* A|B → C than A → C | B → C; or, A → B|C than A → B | A → C). Moreover, such an approach facilitates the automatic re-programming of component definitions in response to the use of similar I/O constraints. The idea is to let the CPU assume more of the selection task by running a specified number of rule alternates against the specified I/O constraints. This off-loads the mundane work to the machine and frees the software engineer in proportion to the processing speed of the machine. Here, the software engineer is freed to work at the conceptual level; while, the machine is enabled to work at the detailed level. Each is liberated to do what it does best. The number of (macro) Boolean components, (macro) procedural components, and alternate candidate rules is determined by the ply of each and the processing speed of the machine. Notice that the task of programming component rules is thus proportionately relaxed. Programming is not necessarily eliminated; rather, it is moved to ever-higher levels. This is randomization [9]. Furthermore, component-type rule-based languages have the advantage of being self-documenting (*e.g*., IF “Root-Problem” THEN “Newton-

Iterative-Method"). Novel and efficient development environments can be designed to support the pragmatics of such programming.

Each run may synthesize semantically equivalent (*i.e.*, within the limits defined by the I/O test vectors), but syntactically distinct functions (*e.g.*, see the alternative definitions for MYSORT at the bottom of Fig. **2**). Similar diversified components are captured in transformation rules. Thus, initially diversified components are synthesized entirely by chance, which of course can be very slow. Chance synthesis is a continual on-going process, which is necessary to maintain genetic diversity. But, once transformation rules are synthesized, they are applied to constituent component rules to create diversified components with great rapidity. The 3-2-1 skew may be applied to favor the use of recently acquired or fired transformation rules. It uses a logical move-to-the-head ordered search based upon temporal locality [21]. The acquisition of new components leads to the acquisition of new transforms. Note that if the system sits idle for long, it enters dream mode *via* the 3-2-1 skew. That is, it progressively incorporates less recently acquired/fired transforms in the search for diversified components.

Transformation rules can be set to minimize space and/or maximize speed and in so doing generalize/optimize. Such optimizations are also in keeping with Occam's razor, which states that in selecting among competing explanations of apparent equal validity, the simplest is to be preferred. If, after each such transformation, the progressively outer components do not properly map their I/O characterization vectors, then it can only be because the pair of components comprising the transformation rule is not semantically equivalent. In this case, the transformation is undone and the transformation rule and its substituted component are expunged (*i.e.*, since it has an unknown deleterious I/O behavior). This allows for a proper version to be subsequently re-synthesized. Components having more-specific redundant rules have those rules expunged.

Convergence upon correct components and thus correct transforms is assured. This is superior to just using multiple analogies as it provides practical (*i.e.*, to the limits of the supplied test vectors) absolute verification at potentially multiple component levels. Such validation is not in contradiction with the Incompleteness Theorem as the test vectors are always finite as is the allowed runtime [17].

4.5. Non Monotonic Rules

Non monotonic rules are secondary rules, which condition the firing of primary rules. They have the advantage of being highly reusable; facilitating the specification of complex components. Reuse is a tenet of randomization theory [13]. Both local and global blackboards utilize posting and retraction protocols. The scope of a local blackboard is limited to the originating component and all components invoked by it. For example,

{Laces: Pull untied laces, Tie: Make bow} → GRETRACT: (Foot-ware: Shoes are untied); GPOST: (Foot-ware: Shoes are tied)

The order of the predefined, global and local RETRACT and POST procedures is, akin to all procedural sequences, immutable.

4.6. Component Redundancy and Diversification

The pattern-matching search known as backtracking can iteratively expand the leftmost node, or the rightmost node on Open [22]. Results here are not identical, but are statistically equivalent. If one component is provided with one expansion search parameter, the other component must be provided with the same search parameter, or the resultant dual-component search will have some breadth-first, rather than strictly depth-first characteristics. This will change the semantics resulting from the use of large search spaces. Clearly, components need to be transformed with due regard for subtle context to preserve their aggregate semantics. These semantic differences become apparent on input vectors, which are outside of those used for I/O definition. Their use can result in erroneous communications *via* the local and/or global blackboards. The system of systems, described in the technical approach below, evolves such context-sensitive components and their transformations.

NNCSs can potentially provide exponentially more security than can a multi-compiler by finding multiple paths from start to goal states [2, 22]. Under syntactic differentiation, achieving the same results implies computing the same component semantics. Under transformational equivalence, one need not compute the same exact component semantics; only ones that achieve the same results in the context of other components. Given sufficiently large problem spaces and sufficient computational power, exponential increases in cyber security can thus be had. Syntactic differentiation can at best provide only linear increases in cybersecurity. Thus, our methodology offers far greater security against

cyberattacks than can conventional approaches [3].

The transformational process converges on the synthesis of syntactically distinct components, which are, to the limits of testing, semantically equivalent. Such components can be verified to be free from attack if their I/O synthesis behavior is within the specified tolerance. Even so, multiple "semantically equivalent" components may compute different output vectors on the same, previously untested input vectors. Here, diversity enables the use of multiple functional analogies by counting the number of diverse components yielding the same output vector. It also allows for a count of the approximate number of recursively enumerable distinct paths leading to the synthesis of each component. This multiple analogies of derivation, when combined with multiple functional analogies, provide a relative validity metric for voting the novel output vectors. These solution vectors are very important because they evidence the system capability for learning to properly generalize by way of exploiting redundancy (*i.e*., in both function and derivation). Furthermore, having multiple derivations provides stochastic non deterministic probabilities. This lies at the root of human imagination and knowledge.

SUMMARY AND CONCLUSION

The goal of this paper was to show that cybersecurity problems can be seen to be a blessing in disguise. They are a blessing because their solution will entail advancing automatic functional programming through KBSE. Non-deterministic synthesized functions allow for the benchmarking of proper behavior; and, this allows for the detection of and recovery from cyberattacks.

KBSE implies the synthesis of code as a last step and the synthesis (generalization) of knowledge as an intermediate step. This implies that cybersecurity is a function of scale because semantic randomization is dependent upon a maximal quantity of most-general knowledge, which is arrived at through spiral development across a network of coherent expert compilers. Such networks are self-amplifying over time. It follows that cybersecurity does not have a static solution. Automation will eventually provide a cure for the cyber-maladies it brought on. Table **1** presents some other traditional methods and randomization-based methods for cybersecurity and recounts their major advantage(s) favoring their use. Unlike all of these methods, semantic randomization allows cybersecurity to follow schema-based automatic programming for cost minimization as a function of scale, as illustrated in Fig. (**2**). Scale cannot be addressed by parallel processors alone. There comes a point where the inclusion of good search control heuristics becomes essential. Semantic randomization has been applied to actual naval problems [23 - 25]. The performance of the system (*i.e.,* the inferential error rate) is tied to the size of the transformational base as well as the processing power/time allocated in conjunction with the schema-definition language. This is unbounded, by any non-trivial metric, because the Kolmogorov complexity of computational intelligence is unbounded [26]. References [27 - 61] contain supporting material, which supplements many of the topics covered in this paper.

Table 1. A comparison of transformation-based cyber security methods.

Number	Method	Diversity Provided
1	Instruction Set Randomization (ISR)	Changes the processor instruction set to foil cyberattacks
2	Address Space Randomization (ASR)	Used by Windows Vista OS to resist memory corruption attacks
3	Data Space Randomization (DSR)	Uses masking to prevent memory corruption better than ASR
4	N-Variant Approaches	If the same input is supplied to a set of diversified variants of the same code, then the cyberattack will succeed on at most one variant.
5	Multi-Variant Code	This technique runs variants of the same program and compares the behavior of the variants at synchronization points
6	Behavioral Distance	A way to defend against mimicry attacks by using a comparison between the behaviors of two diverse processes running the same input.
7	Semantic Randomization	A way to apply schema-based program synthesis to automatically create semantically equivalent syntactically distinct algorithmic variants of the same code, where a cyberattack will succeed on at most one variant. Picks up where the use of distinct compilers leaves off. May be used in combination with the above methods for enhanced cybersecurity.

CONSENT FOR PUBLICATION

Not applicable.

CONFLICT OF INTEREST

The authors declare no conflict of interest, financial or otherwise.

ACKNOWLEGEMENTS

Dr. Rubin would like extend his thanks to Dr. Sukarno Mertoguno of the Office of Naval Research (ONR) and Dr. Matthew Stafford of the U.S. Air Force (USAF) for supporting this research. This work was produced by a U.S. government employee as part of his official duties and is not subject to copyright. It is approved for public release with an unlimited distribution.

REFERENCES

[1] A.J. Kfoury, R.N. Moll, and M.A. Arbib, *A Programming approach to computability.,* Springer-Verlag Inc.: New York, NY, 1982. [http://dx.doi.org/10.1007/978-1-4612-5749-3]

[2] S. Singh, and S. Silakari, "A survey of cyber attack detection systems", *Int. J. of Computer Science and Network Security,* vol. 9, no. 5, pp. 1-10, 2009. [IJCSNS].

[3] A. Gherbi, R. Charpentier, and M. Couture, "Software diversity for future systems security", *Workshop on Hot Topics in Operating Systems,* pp. 10pp. 57-13-72, 2011.

[4] "Computer security, dependability, and assurance: From needs to solutions: Proceedings 7-9 Jul. 1998, York, England, 11-13 Nov", 1998, Williamsburg, VA, IEEE Computer Society Press.

[5] J. Hindin, "Intelligent tools automate high-level language programming", *Comput. Des. (Winchester),* vol. 25, pp. 45-56, 1986.

[6] A.W. Biermann, "Automatic programming, A tutorial on formal methodologies", *J. Symbolic Comp.,* vol. 1, no. 2, 1985. [http://dx.doi.org/10.1016/S0747-7171(85)80010-9]

[7] E.A. Feigenbaum, and P. McCorduck, *The fifth generation: Artificial intelligence and Japan's computer challenge to the world.,* Addison-Wesley Pub. Co.: Reading, MA, 1983.

[8] S. Minton, *Learning search control knowledge: An explanation based approach.,* Kluwer International Series in Engineering and Computer Science: New York, 1988, p. 61. [http://dx.doi.org/10.1007/978-1-4613-1703-6]

[9] G.J. Chaitin, "Randomness and mathematical proof", *Sci. Am.,* vol. 232, no. 5, pp. 47-52, 1975. [http://dx.doi.org/10.1038/scientificamerican0575-47]

[10] H. Hosseini, and R. Poovendran, Deep neural networks do not recognize negative images, work was supported by ONR grants N00014-14-1-0029 and N00014-16-1-2710, ARO grant W911NF-16-1-0485, and NSF grant CNS-1446866, pp. 1-3, 2017.

[11] H. Hosseini, Deceiving google's perspective API built for detecting toxic comments, work was supported by ONR grants N00014-14-1-0029 and N00014-16-1-2710 and ARO grant W911NF-16-1-0485, pp. 1-4, 2017.

[12] S. Amarel, "On representations of problems of reasoning about actions", *Math. Intell.,* vol. 3, pp. 131-171, 1968.

[13] S.H. Rubin, "On randomization and discovery", *Inf. Sci.,* vol. 177, no. 1, pp. 170-191, 2007. [http://dx.doi.org/10.1016/j.ins.2006.06.001]

[14] R. Solomonoff, "A new method for discovering the grammars of phrase structure languages", *Proc. Int. Conf. Information Processing,* , 1959pp. 285-290

[15] R. Solomonoff, "A formal theory of inductive inference, inform", In: *Contr.,* 1964. pp. 7 1-22 and 224-254

[16] M.A. Arbib, A.J. Kfoury, and R.N. Moll, *A basis for theoretical computer science.,* Springer-Verlag: New York, NY, 1981. [http://dx.doi.org/10.1007/978-1-4613-9455-6]

[17] V.A. Uspenskii, *Gödel's incompleteness theorem, translated from Russian.,* Ves Mir Publishers: Moscow, 1987.

[18] Q. Liang, and S.H. Rubin, "Randomization for testing systems of systems", *Proc. 10th IEEE Intern. Conf.,* , 2009pp. 110-114 [http://dx.doi.org/10.1109/IRI.2009.5211597]

[19] A.N. Kolmogorov, and V.A. Uspenskii, "On the definition of an algorithm", In: *Russian, English Translation: Amer. Math. Soc. Translation,* vol. 29 . 1963, no. 2, pp. 217-245. [http://dx.doi.org/10.1090/trans2/029/07]

[20] T. Bouabana-Tebibel, S.H. Rubin, Eds., *Integration of reusable systems.,* Springer International Publishing: Switzerland, 2014. [http://dx.doi.org/10.1007/978-3-319-04717-1]

[21] H.M. Deitel, *An introduction to operating systems.,* Prentice Hall, Inc.: Upper Saddle River, NJ, 1984.

[22] N.J. Nilsson, *Principles of artificial intelligence.,* Tioga Pub. Co.: Palo Alto, CA, 1980.

[23] S.H. Rubin, G. Lee, and S.C. Chen, "A case-based reasoning decision support system for fleet maintenance", In: *Naval Engineers J.*. NEJ-2009-05-STP-0239.R1, pp. 1-10, 2009.

[24] S.H. Rubin, Multi-level segmented case-based learning systems for multi-sensor fusion. NC No. 101451, 2011.

[25] S.H. Rubin, Multilevel constraint-based randomization adapting case-based learning to fuse sensor data for autonomous predictive analysis. NC 101614, 2012.

[26] S.H. Rubin, "Is the kolmogorov complexity of computational intelligence bounded above?", In: *12th IEEE Intern. Conf. Info. Reuse & Integration, Las Vegas,* 2011, pp. 455-461.
[http://dx.doi.org/10.1109/IRI.2011.6009591]

[27] V. Behbood, J. Lu, and G. Zhang, "Text categorization by fuzzy domain adaptation", *IEEE International Conference on Fuzzy Systems,* 2013 Hyderabad, India

[28] D. Song, M. Reiter, and S. Forrest, Taking cues from mother nature to foil cyber attacks, NSF PR 03-130, https:// www.nsf. gov/od/ lpa/news/ 03/pr031.

[29] S. Forrest, A. Somayaji, and D.H. Ackley, "Building diverse computer systems", *Workshop on hot topics in operating systems,* 1997pp. 57-72

[30] G.P.C. Fung, J.X. Yu, H.J. Lu, and P.S. Yu, "Text classification without negative examples revisit", *IEEE Trans. Knowl. Data Eng.,* vol. 18, no. 1, pp. 6-20, 2006.
[http://dx.doi.org/10.1109/TKDE.2006.16]

[31] S.J. Pan, and Q. Yang, "A survey on transfer learning", *IEEE Trans. Knowl. Data Eng.,* vol. 22, no. 10, pp. 1345-1359, 2010.
[http://dx.doi.org/10.1109/TKDE.2009.191]

[32] J. Lu, V. Behbood, P. Hao, H. Zuo, S. Xue, and G. Zhang, "Transfer learning using computational intelligence: A survey", *Knowl. Base. Syst.,* vol. 80, pp. 14-23, 2015.
[http://dx.doi.org/10.1016/j.knosys.2015.01.010]

[33] J-T. Huang, J. Li, D. Yu, L. Deng, and Y. Gong, "Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers", *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP),* 2013 Vancouver, Canada
[http://dx.doi.org/10.1109/ICASSP.2013.6639081]

[34] P. Swietojanski, A. Ghoshal, and S. Renals, Unsupervised cross-lingual knowledge transfer in DNN-based LVCSR.*IEEE Wkshp on Spoken Lang.,* Tech: Miami, USA, 2012.
[http://dx.doi.org/10.1109/SLT.2012.6424230]

[35] V. Behbood, J. Lu, and G. Zhang, "Text categorization by fuzzy domain adaptation", *IEEE International Conference on Fuzzy Systems,* 2013 Hyderabad, India

[36] D.C. Ciresan, U. Meier, and J. Schmidhuber, "Transfer learning for Latin and Chinese characters with deep neural networks", *Int. Joint Conf. on Neural Networks,* 2012 Australia
[http://dx.doi.org/10.1109/IJCNN.2012.6252544]

[37] C. Kandaswamy, L.M. Silva, L.A. Alexandre, J.M. Santos, and J.M. De Sa, "Improving deep neural network performance by reusing features trained with transductive transference", *24th Int. Conf. on Artificial Neural Networks,* 2014 Hamburg, Germany
[http://dx.doi.org/10.1007/978-3-319-11179-7_34]

[38] J. Shell, and S. Coupland, Towards fuzzy transfer learning for intelligent environments.*Ambient Intelligence,* 2012, pp. 145-160.
[http://dx.doi.org/10.1007/978-3-642-34898-3_10]

[39] L.A. Celiberto Jr, J.P. Matsuura, R.L. De Mantaras, and R.A.C. Bianchi, "Using cases as heuristics in reinforcement learning: A transfer learning application", *Int. Joint Conf. on Artificial Intelligence,* 2011 Barcelona, Spain

[40] A. Niculescu-Mizil, and R. Caruana, "Inductive transfer for Bayesian network structure learning", *11th Int. Conf. on Artificial Intelligence and Statistics,* 2007 Puerto Rico

[41] D. Oyen, and T. Lane, "Bayesian discovery of multiple Bayesian networks *via* transfer learning", *13th IEEE International Conference on Data Mining (ICDM),* 2013 Dalla, USA
[http://dx.doi.org/10.1109/ICDM.2013.90]

[42] V. Behbood, J. Lu, and G. Zhang, "Long term bank failure prediction using fuzzy refinement-based transductive transfer learning", *IEEE Int. Conf. on Fuzzy Systems,* 2011 Taiwan
[http://dx.doi.org/10.1109/FUZZY.2011.6007633]

[43] V. Behbood, J. Lu, and G. Zhang, "Fuzzy bridged refinement domain adaptation: Long-term bank failure prediction", *Int. J. Comput. Intell. Appl.,* vol. 12, no. 01, 2013.

[44] V. Behbood, J. Lu, and G. Zhang, "Fuzzy refinement domain adaptation for long term prediction in banking ecosystem", *IEEE Trans. Industr. Inform.,* vol. 10, no. 2, pp. 1637-1646, 2014.
[http://dx.doi.org/10.1109/TII.2012.2232935]

[45] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction", *Inf. Softw. Technol.,* vol. 54, no. 3, pp. 248-256, 2012.

[http://dx.doi.org/10.1016/j.infsof.2011.09.007]

[46] R. Luis, L.E. Sucar, and E.F. Morales, "Inductive transfer for learning Bayesian networks", *Mach. Learn.,* vol. 79, no. 1–2, pp. 227-255, 2010. [http://dx.doi.org/10.1007/s10994-009-5160-4]

[47] J. Shell, Fuzzy transfer learning*,* Ph.D. Thesis, De Montfort University, 2013.

[48] S. Chopra, S. Balakrishnan, and R. Gopalan, "DLID: deep learning for domain adaptation by interpolating between domains", *ICML Workshop on Challenges in Representation Learning,* 2013 Atlanta, USA

[49] R. Caruana, "Multitask learning: A knowledge-based source of inductive bias", *Tenth International Conference of Machine Learning,* 1993 MA, USA
[http://dx.doi.org/10.1016/B978-1-55860-307-3.50012-5]

[50] R. Caruana, "Multitask learning", *Mach. Learn.,* vol. 28, pp. 41-75, 1997.
[http://dx.doi.org/10.1023/A:1007379606734]

[51] D.L. Silver, and R. Poirier, "Context-sensitive MTL networks for machine lifelong learning", *20th Florida Artificial Intelligence Research Society Conference,* 2007 Key West, USA

[52] W. Dai, G. Xue, Q. Yang, and Y. Yu, "Transferring naive Bayes classifiers for text classification", *22nd National Conference on Artificial Intelligence,* 2007 Vancouver, Canada

[53] D.M. Roy, and L.P. Kaelbling, "Efficient Bayesian task-level transfer learning", *International Joint Conference on Artificial Intelligence,* 2007 Hyderabad, India

[54] J. Shell, and S. Coupland, "Fuzzy transfer learning: Methodology and application", *Inf. Sci.,* vol. 293, pp. 59-79, 2015.
[http://dx.doi.org/10.1016/j.ins.2014.09.004]

[55] B. Koçer, and A. Arslan, "Genetic transfer learning", *Expert Syst. Appl.,* vol. 37, no. 10, pp. 6997-7002, 2010.
[http://dx.doi.org/10.1016/j.eswa.2010.03.019]

[56] V. Honavar, G. Slutzki, Eds., *Grammatical inference, lecture notes in artificial intelligence 1433.,* Springer-Verlag: Berlin, Germany, 1998.

[57] K.S. Fu, *Syntactic pattern recognition and applications.,* Prentice-Hall Advances in Computing Science and Technology Series: Englewood Cliffs, NJ, 1982.

[58] S.H. Rubin, "Computing with words", *IEEE Trans. Syst., Man, and Cybern. Part B,* vol. 29, no. 4, pp. 518-524, 1999.

[59] S.H. Rubin, G. Lee, and S.C. Chen, " A case-based reasoning decision support system for fleet maintenance", , *Naval Engineers J.,* 2009. NEJ-2009-05-STP-0239.R1, pp. 1-10, 2009.

[60] S.H. Rubin, Multi-level segmented case-based learning systems for multi-sensor fusion, NC No. 101451, 2011.

[61] S.H. Rubin, Multilevel constraint-based randomization adapting case-based learning to fuse sensor data for autonomous predictive analysis. NC 101614, 2012.