UnProVET: Using Explicit Constraint Propagation to Construct Attack Graphs

Dmitry Gorbatenko, Alexander Semenov, Stepan Kochemazov
Matrosov Institute for System Dynamics and Control Theory SB RAS, Irkutsk, Russia
Email: gorbadima@yandex.ru, biclop.rambler@yandex.ru, veinamond@gmail.com

***Abstract*—One of the important problems in network security consists in the construction and analysis of attack graphs which represent all possible attacks a malefactor can carry out within a specific computer network. In this paper we describe the software system for constructing such attack graphs. The system is based on the constraint propagation mechanisms which are very close to that employed by the algorithms for solving Constraint Satisfaction Problem (CSP) and its variants. Unlike several other known software systems for attack graph generation it employs explicit constraint propagation implemented using special data structures. The computational experiments show that the system has good performance and outperforms the competition in certain scenarios.**

I. Introduction

Attack graphs are the special labeled graphs used to represent the development of attacks in computer networks. They are an important object of the study in both theoretical and practical areas of computer security. There are multiple ways of representing how the attacks develop in the network using special graphs. The first representations of this kind were very ineffective [1]. In [2] there was described the first class of attack graphs, for which the time required to generate them grows as a polynom on the number of hosts in a corresponding computer network. After [2] there have been published a number of papers which proposed the classes of attack graphs with steadily declining complexity of their generation. We would like to specifically point out the papers [3], [4], where there have been proposed the class of attack graphs with practical effectiveness of their construction lying between $O(n^2)$ and $O(n^3)$. These papers are based on the idea to synthesize an attack graph during solving the logical programming problem: essentially, the attack graph is the output of some Prolog program. The main disadvantage of the attack graphs from [3], [4] is that they have loops, which significantly hinder correct interpretation of attacks. Also, in order to outline the set of shortest attacks in a graph with loops one has to perform a lot of laborious computations. To the best of our knowledge, today the most compact attack graphs are the ones constructed by the approach described in [5], [6], [7]. However, the corresponding graphs too contain loops. As it is mentioned in the review [8], the `NetSPA` system later was included into the commercial tool which is not available for public use. Therefore, we could not compare our results with that of `NetSPA`. Note, that a promising trend in attack graph generation is captured in e.g. [9], where it was proposed to use distributed computing for this purpose. The current implementation of our tool is a single-threaded program, but in the nearest future we plan to consider the parallelization of the algorithms we employ. An additional information on the research related to attack graphs and their generation can be found in the review [10].

In [11], [12] we proposed a new class of attack graphs which are generated as a result of a discrete dynamical process of automaton type. The resulting attack graphs do not contain loops and the development of each attack has strict time hierarchy. The complexity of constructing such attack graph is $O(n^2)$. The procedure that finds the shortest attacks in a network has the same complexity. Also in [11], [12] we described the propositional encoding technique for the algorithms that construct graphs of this class. The result of such encoding is a Boolean formula in Conjunctive Normal Form (CNF). This formula can be used to solve various combinatorial problems aimed at figuring out the security properties of a considered network, for example, to distribute patches that deactivate some vulnerabilities on specific hosts. This formula can be used to construct an attack graph by iteratively applying the Unit Propagation rule to it.

In this paper we present the new variant of the software system used in [11], [12], which employs explicit constraint propagation to generate attack graphs. The new version works directly with data structures containing the information about a network without the transition to Boolean expressions. This approach is justified in the case when one wants to construct an attack graph fast, and there is no need to solve any related combinatorial problems. We show that the speed of attack graphs generation achieved by this new procedure significantly exceeds that of the generation via Boolean constraint propagation.

Let us give a brief outline of our paper. In the next section we give the main notation and results used as a basis for the following constructions. In particular, the section contains a short description of the UnProVET system. In Section 3 we describe the new variant of UnProVET in which the constraint propagation is implemented without derivation mechanisms from Boolean logic. The fourth section contains the results of computational experiments on comparison of UnProVET with similar systems.

II. Basic notation and Background

Below let us consider the process of attack development in a computer network as a discrete dynamical process of

automaton type. In our formal description we use the notation system introduced in [11], [12].

Hereinafter, we use an undirected labeled graph $G = (V, E, L_V)$ as a model of computer network, where $V \to L_V$ is the mapping that defines the vertices' labels: each label contains some information about a vertex. The set V, $|V| = n$ is associated with a set of *hosts* of a computer network, and each edge $e = (a, b), e \in E$ interprets the physical connection between hosts $a, b \in V$. A *neighborhood* of an arbitrary vertex $v \in V$ is the set of vertices adjacent to v. Assume that there are two specifically outlined vertices in V: v_M and v_T. Here v_M is the vertex which represents the host of a malefactor. The goal of a malefactor is to obtain control over host v_T.

An important assumption is that there is defined a mapping $V \to L_V$ which associates with an arbitrary vertex $v \in V$ some information (label). In particular, with each host $v \in V$ there is linked a binary word l_v. Every coordinate of l_v shows whether or not a host v has a particular vulnerability. We refer to l_v as *vector of vulnerabilities*. Informally, the vulnerabilities are the conditions which make it possible for the malefactor attacking v to escalate its access rights on v. Let us denote by U the list of all possible vulnerabilities that can appear on network hosts. Then for an arbitrary v the vector l_v will contain $|U|$ numbers from $\{0, 1\}$, the 1-coordinates signifying that the corresponding vulnerabilities from U are present at v.

In the context of the model employed in [11], [12] we assume that the only active subject in a network is the malefactor. Its actions take place at discrete time moments and consist in escalating malefactor's privileges on hosts available to the malefactor at this time moment. In [11], [12] there have been considered the following access levels:

1) Access level $user_0$ of host v' to host v'' means that v' can read $l_{v''}$, i.e. v' (which is either v_M or a host on which M has high access rights) can physically connect to v''.
2) Access level $user$ of host v' to host v'' means that v' can perform elementary attacks on v''.
3) Access level $root$ of host v' to host v'' means that v' gains the access level $user_0$ to all hosts from the neighborhood of v'' in graph G.

Informally, the elementary attack mentioned in point 2 is the atomic action of a malefactor which consists in exploitation of one or several vulnerabilities of a considered host.

The introduced gradation of access levels makes it possible to consider the development of attacks in computer network as a dynamic process taking place at discrete time moments. Below we go in a little bit more detail regarding the way the development of attacks was represented in [12]. The corresponding conditions on execution of elementary attacks were formulated as close to the ones in [3] as possible. The successful execution of any attack (including elementary attacks) is the escalation of current access rights of M to the next possible level in the gradation above. The attack on a network is deemed to be successful if at some time moment M manages to get $root$ access to host v_T.

Thus, assume that at $t = 0$ the malefactor can access only the hosts with which it is physically connected, i.e. is in the relation $connect$ with them. Then at time moment $t = 1$ the access level of M to these hosts is elevated to $user_0$. If the host to which M has access rights $user_0$ possesses the *remote_exploit* vulnerability then at the next time moment M escalates its access rights to this host to $user$. If M has $user$ rights on some host on which there exists the *local_exploit* vulnerability then M can get the $root$ access to the corresponding host at the next time moment. Finally, if M has the $root$ access to host h at time moment t, then at $t + 1$ it is assumed to be in the $connect$ relation with all hosts from the neighborhood $N(h)$.

In [11], [12] it was shown that the process of development of an attack in computer network corresponds to the State Transition Graph (STG) Γ_G of a discrete dynamical system (DDS) which we denote as Δ_G. The graph of attacks on network G essentially is a special representation of Γ_G. Also in these papers it was shown that under the monotonicity assumption over the discrete automaton mapping represented by graph Γ_G, the DDS Δ_G has a single stationary point and does not have cycles of length > 1. Thus it is easy to make the conclusion that the attack graph A_G constructed from Γ_G is a directed graph without loops (unlike the graphs generated by the MULVAL system [3]). The complexity of constructing A_G for G (under standard assumptions for computer security) is $O(n^2)$, where n is the number of hosts in network G.

III. Attack Graph Generation as the Constraint Propagation Process

In [12] we presented the UNPROVET (Unit Propagation Escalation Vulnerability Tool) system for generating attack graphs. In this system an algorithm that constructs A_G based on Γ_G is encoded into a Boolean formula C_G in a Conjunctive Normal Form (CNF). For this purpose the tool employs the standard symbolic execution mechanisms described for example in [13]. CNF C_G can be used to solve various combinatorial problems related to A_G. For example, it can be the patching problem: to patch a relatively small number of vulnerabilities on some network hosts in such a way that all previously possible attacks on target host become impossible. The information about the network which does not change with time, and also the initial state of the malefactor are represented via the unit clauses which are conjunctively added to C_G. Then the tool applies to an obtained CNF the simplest Boolean Constraint Propagation rule called the Unit Propagation Rule [14], [15]. The result of its iterative use is the derivation of the values for all variables in C_G. These values are then used to effectively construct an attack graph A_G. In the present paper we describe the variant of UNPROVET in which the constraint propagation is implemented directly over data structures used in the transition from G to Γ_G and A_G without constructing CNF C_G.

Below we give examples to key notions mentioned above. The example of a graph G representing a computer network is given on Figure 1.

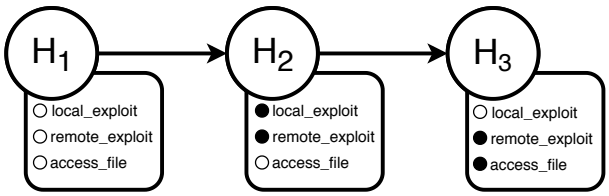

Fig. 1. The example of the network graph with three hosts and their vulnerabilities.

Let us comment on Figure 1. It shows the network with three hosts (H_1, H_2, H_3). There are three kinds of vulnerabilities: *local_exploit*, *remote_exploit*, *access_file*. The *remote_exploit* vulnerability allows the malefactor to elevate its access rights from $user_0$ to $user$. The *local_exploit* vulnerability makes it possible to improve it from $user$ to $root$. Finally, the *access_file* vulnerability gives the malefactor access to file system and thus to use malicious software. Hereinafter we assume that by exploiting it the malefactor launches some trojan.

The exploitation of one vulnerability is called an elementary attack. In the context of the approach used in [11], [12] with each separate realization of elementary attack we associate two discrete time moments: t and $t+1$. In more detail, we assume that at t the conditions on this elementary attack are satisfied (we refer to them as *preconditions*), and at moment $t+1$ we observe the result of this attack (*postcondition*). The elementary attacks most often considered in practice can be split into the following two classes (here we follow [3]):

- *NetAccess* corresponds to determining if there is physical interconnection between hosts. It is responsible for obtaining the access rights $user_0$.
- *ExecCode* corresponds to successful execution of malicious software on an attacked host. It is responsible for escalation of access rights from $user_0$ to $user$ and from $user$ to $root$.

In Tables I and II these attacks are represented by pairs (precondition, postcondition).

Attack name	multi-hopAccess	directNetworkAccess
Preconditions	$root^S$ $connect_S^T$	$root^M$ $connect_M^T$
Postcondition	$user_0^T$	$user_0^T$

TABLE I
ATTACKS FROM $NetAccess$ CLASS

Attack name	remoteExploit	localExploit	trojanInstall
Preconditions	$user_0^T$ *remote_exploit*^{T}	$user^T$ *local_exploit*^{T}	$user^T$ *access_file*^{T}
Postcondition	$user^T$	$root^T$	$root^T$

TABLE II
ATTACKS FROM $ExecCode$ CLASS

Here by S (source) and T (target) we denote the host from which M starts the attack and the attacked host. The access level of a malefactor on a particular host is denoted by superscript (for example, $user^T$ means that M has $user$ rights on host T).

The condition $connect_K^L$ is a predicate which takes the value of $True$ if and only if hosts K and L are connected by an edge in G (i.e. there is direct physical connection between these two hosts).

Thus, in order to perform the *remoteExploit* attack on host T the malefactor must gain $user_0$ on this host at the preceding time moments. To do this M has to get $root$ access to one of the hosts physically connected to T. Also, T must have the *remote_exploit* vulnerability.

An arbitrary state of a DDS Δ_G defined by network G is (in accordance with [11]) a binary matrix of size $3 \times n$: the element (i, j) of this matrix is equal to 1 if and only if at the corresponding time moment the malefactor M has access level number j to the host number i, $i \in \{1, \ldots, n\}$, where $j \in \{1, 2, 3\}$, $1 = user_0$, $2 = user$, $3 = root$.

As it was noted above, the DDS Δ_G does not have cycles of length ≥ 2 and has exactly one stationary point. An arbitrary transition from t to $t+1$ corresponds to changing some components in the state matrix from 0 to 1. Taking into account the standard for computer security assumption on the monotonicity of malefactor's actions (i.e. the results of elementary attacks on preceding steps can not be cancelled later), it is easy to see that the number of different states of Δ_G does not exceed $3n$. All possible transitions between states of Δ_G (i.e. the State Transition Graph, STG Γ_G) can be represented by a table. The Table III specifies the STG for the network from Figure 1.

Time moment t	DDS state Δ_G	Elementary attack corresponding to transition $t \to t+1$
0	1 1 1 0 0 0 0 0 0	
1	1 1 1 1 0 0 0 0 0	DIRECTNETWORKACCESS(H_2)
2	1 1 1 1 1 0 0 0 0	REMOTEEXPLOIT(H_2)
3	1 1 1 1 1 1 0 0 0	LOCALEXPLOIT(H_2)
5	1 1 1 1 1 1 1 0 0	MULTI-HOPACCESS(H_2,H_3)
5	1 1 1 1 1 1 1 1 0	REMOTEEXPLOIT(H_2)
6	1 1 1 1 1 1 1 1 1	TROJANINSTALL(H_3)

TABLE III
ALL POSSIBLE TRANSITIONS BETWEEN STATES OF DDS Δ_G FOR THE NETWORK FROM FIGURE 1.

Comments to Table III. From the table it follows that Δ_G

for the network from Figure 1 has 6 states. The initial state corresponds to the situation when M controls only its own host H_1. The final state represents the situation when M launches a trojan on host H_3. After this point there are no possible actions in the system (so it is a stationary point).

IV. The UnProVET system

The UnProVET software system was designed to solve a variety of problems related to attack graph generation. It was implemented in C++. Similar to MulVAL, the UnProVET system takes as an input an .xml file which contains the description of a network, of vulnerabilities on the hosts, of *and*- and *or*-attacks. After the file is loaded the user can use the graphical interface to edit scenarios, e.g. add or remove facts, modify, add or delete attacks, etc. After this the tool can be used to generate an attack graph. This can be done using two distinctive approaches: the first involves reducing the problem of attack graph generation to the Boolean satisfiability problem (SAT) [16] and using SAT solvers to generate them [12]. The second approach implies direct constraint propagation via the corresponding C++ implementation. The attack graph can be visualized using Graphviz [17]. The result is similar to the one presented at Figure 2.

As it was mentioned above, the UnProVET system essentially performs the transition from Γ_G to an attack graph in its traditional sense. In particular it uses as its basis the format for attack graphs used by the MulVAL system [3]. Such attack graphs contain vertices of three kinds:

- The rectangular vertices in attack graph show the information related to a state of a network, which can not change with time. For example, such information can be about the physical connection between two hosts or the list of installed programs on each host. Let us join all such information into the set to which we refer as set of facts and denote by F.
- The oval vertices represent elementary attacks. The vertices from which the arcs go into oval vertex mean the preconditions of this attack. The vertex to which the edge goes from the oval vertex interprets the postcondition of an elementary attack. Let us denote the set of elementary attacks as A and call it the set of "and"-attacks, since their realization requires that all preconditions are present.
- The diamond-shaped vertices interpret the postconditions of elementary attacks. At each time moment the malefactor can theoretically perform several elementary attacks from a single class, which result in the same gain (i.e. several elementary attacks have the same graph vertex as postcondition). Let us denote the set of diamond-shaped vertices by O and refer to it as the set of "or"-attacks, since their realization implies that at least one of related individual "and"-attacks is realized.
- The arcs of graph A_G represent the transitions between time moments. Denote the set of arcs as E.

Thus the attack graph A_G is the quadruplet of sets $A_G = (F, O, A, E)$. It is a directed acyclic graph the vertices of which can formally be divided into Q layers, where Q is the number of distinctive states in Δ_G. Thus, each layer corresponds to a particular time moment.

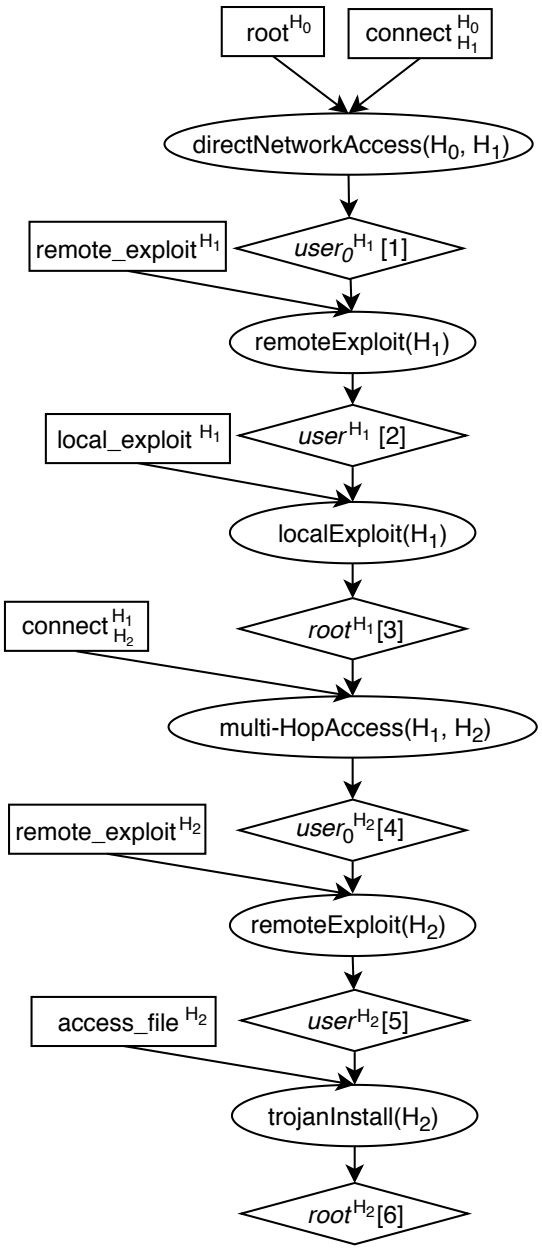

Fig. 2. The example of attack graph for the network from Figure 1

Initially in UnProVET all elementary steps that take place in DDS Δ_G from $t = 0$ and until the system enters the stationary point were encoded into Boolean formulas which in turn were joined into a general formula C_F in Conjunctive Normal Form. With each fact there was associated some literal. All literals were conjunctively added to C_G. Then the Unit Propagation rule [14], [15] was iteratively applied to the obtained CNF $\tilde{C}_G$. This resulted in derivation of values for all variables from $\tilde{C}_F$ and thus provided the information about all possible attacks in network G. This information can be effectively transformed into attack graph A_G.

Let us briefly describe another possible approach to construction of A_G based on graph Γ_G (assuming that it is defined by a table similar to Table III). In the context of this approach the constraint propagation procedure is realized directly over the data structures containing the information about graphs G and Γ_G. In [12] it was shown that the knowledge of facts (i.e. vulnerabilities, interconnections between hosts, etc.) is sufficient for constructing an attack graph. However, it is possible to do it without Unit Propagation rule. Thus, in this paper we implemented direct constraint propagation over the data. It means that essentially the system starts from the initial state (of a DDS, see e.g. Table III) and constructs the next state by applying all possible transformation rules (i.e. elementary attacks) to a present state. The program stops its work after the two consecutive states are equal (i.e. the system enters a stationary point). Essentially, the SAT variant from [12] and the direct implementation perform the exact same work but by different means.

V. Computational Experiments

In the computational experiments we compare UNPROVET with the well known MULVAL tool. In particular, we measure the time of attack graph generation by MULVAL with that of two implementations in UNPROVET: the one employing derivation in Boolean logic and the one that uses explicit constraint propagation directly over available data.

In the experiments we used the networks generated according to the following algorithm. To represent networks we use adjacency matrices. For a given network size N, the adjacency matrix is initially filled with all zeroes. Then starting from vertex number 1 and up to vertex N first the algorithm randomly chooses the number of edges k_i that will go from vertex v_i randomly from $(3; \frac{N-i}{8})$. If $\frac{N-i}{8} < 1$ then $k_i = 0$. Then the k_i edges are generated between v_i and the vertices randomly chosen from $\{v_{i+1}, \ldots, v_N\}$. Once the network structure is fixed, 75% vertices are randomly chosen and assigned some vulnerability that can theoretically be exploited. The malefactor is always associated with vertex v_1. The goal for all systems is to construct the attack graph representing all possible attacks on a network.

The experiments were performed on one core of Intel Core i7-8700 processor with 16 Gb RAM. The results are presented in Table IV. For each graph dimension 10 networks were generated, thus the data in the Table IV is averaged for 10 tests. The time is presented in seconds.

Dimension	MULVAL	UNPROVET UP	UNPROVET direct
500	3.03	1.16	1.07
1000	9.16	3.2	2.79
2000	68.18	10.6	8.77
5000	1534.51	46.22	35.57

TABLE IV
TIME OF ATTACK GRAPH GENERATION FOR UNPROVET AND MULVAL (IN SECONDS).

The "UNPROVET UP" column corresponds to the variant when the Unit Propagation rule is applied to the corresponding Boolean formulas to generate attack graphs while "UNPROVET direct" shows the result for the proposed direct scheme of constraint propagation. It is clear that in the considered experiments the attack graph generation speed of UNPROVET is considerably higher than that for MULVAL. Note that the direct propagation is always better than its implementation via Boolean logic, but the difference in runtime is always within 30%.

VI. Conclusions and Future Work

In the present paper we described the UNPROVET software system designed for constructing graphs representing possible attacks on computer networks. The novel functionality of UNPROVET which uses explicit constraint propagation directly over C++ data structures shows significantly better performance in attack graphs construction compared to the approach employing the reduction to SAT and using Unit Propagation rule. We also show that UNPROVET works faster than the well known MULVAL system for attack graphs generation.

In the nearest future we plan to extend the functional capabilities of UNPROVET towards being able to solve problems of distributing patches aimed at blocking possible attacks. For this purpose we plan to use the state-of-the-art SAT solvers.

Acknowledgments

Authors thank anonymous reviewers for their valuable comments and suggestions.

The research was funded by Russian Science Foundation (project No. 16-11-10046).

References

[1] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing, "Automated generation and analysis of attack graphs," in *Proceedings 2002 IEEE Symposium on Security and Privacy*, 2002, pp. 273–284.

[2] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, ser. CCS '02, 2002, pp. 217–224.

[3] X. Ou, S. Govindavajhala, and A. W. Appel, "Mulval: A logic-based network security analyzer," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, ser. SSYM'05, 2005, pp. 8–8.

[4] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06, 2006, pp. 336–345.

[5] K. Ingols, R. Lippmann, and K. Piwowarski, "Practical attack graph generation for network defense," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006, pp. 121–130.

[6] K. Ingols, M. Chu, R. Lippmann, S. Webster, and S. Boyer, "Modeling modern network attacks and countermeasures using attack graphs," in *2009 Annual Computer Security Applications Conference*, 2009, pp. 117–126.

[7] M. L. Artz, "NetSPA : a Network Security Planning Architecture," Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science., USA, 2002.

[8] S. Yi, Y. Peng, Q. Xiong, T. Wang, Z. Dai, H. Gao, J. Xu, J. Wang, and L. Xu, "Overview on attack graph generation and visualization technology," in *2013 International Conference on Anti-Counterfeiting, Security and Identification (ASID)*, 2013, pp. 1–6.

[9] K. Kaynar and F. Sivrikaya, "Distributed attack graph generation," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 5, pp. 519–532, 2016.

[10] M. Barik, A. Sengupta, and C. Mazumdar, "Attack graph generation and analysis techniques," *Defence Science Journal*, vol. 66, no. 6, pp. 559–567, 2016.
[11] A. Semenov, D. Gorbatenko, and S. Kochemazov, "Computational study of activation dynamics on networks of arbitrary structure," in *Computational Aspects and Applications in Large-Scale Networks*, ser. Springer Proceedings in Mathematics & Statistics, vol. 247, 2018, pp. 205–220.
[12] D. Gorbatenko and A. Semenov, "Modeling attacks in computer networks using boolean constraint propagation," in *2018 Global Smart Industry Conference (GloSIC)*, 2018, pp. 1–6.
[13] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, vol. 2988. Springer, 2004, pp. 168–176.
[14] W. F. Dowling and J. H. Gallier, "Linear-time algorithms for testing the satisfiability of propositional horn formulae," *J. Log. Program.*, vol. 1, pp. 267–284, 1984.
[15] J. P. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 185, pp. 131–153.
[16] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, 2009.
[17] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *SOFTWARE - PRACTICE AND EXPERIENCE*, vol. 30, no. 11, pp. 1203–1233, 2000.