Finding an Optimal Set of Static Analyzers To Detect Software Vulnerabilities

Jiaqi He, Revan MacQueen, Natalie Bombardieri, Karim Ali, James R. Wright
University of Alberta
{jhe15, revan, nbombard, karim.ali, jwright4}@ualberta.ca

Cristina Cifuentes
Oracle
cristina.cifuentes@oracle.com

2023 IEEE International Conference on Software Maintenance and Evolution (ICSME) | 979-8-3503-2783-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSME58846.2023.00060

***Abstract*—Software vulnerabilities are ubiquitous and costly. To detect vulnerabilities earlier during development, organizations deploy a set of static analyzers to locate and eventually fix these vulnerabilities before releasing their software. Due to the prohibitive cost of running all available analyzers, organizations must run only a subset of all possible analyzers on their codebases. Choosing this set deterministically leaves recognizable gaps of vulnerability coverage. To overcome these challenges, we present Randomized Best Response (RBR), a method that computes an optimal randomization over size-bounded sets of available static analyzers. RBR models the relationship between malicious users and organizations as a leader-follower Stackelberg security game. Our solution focuses on software vulnerabilities due to their security implications when exploited by malicious users. Using 8 static analyzers for C/C++ and 8 Common Weakness Enumeration (CWE) vulnerability types, we show that RBR outperforms a set of natural baselines by always picking analyzers that achieve a higher benefit to the defender. Through a case study of a large system at Oracle, we show how RBR may be used in a real-world scenario.**

I. Introduction

Software vulnerabilities are ubiquitous and costly. According to The Consortium for IT Software Quality (CISQ), poor software quality cost the US economy 2.08 trillion dollars in 2020 [1]. Poorly designed software enables malicious users to exploit vulnerabilities, which are bugs with security implications, within it to perform various types of attacks. To control the damage of potential attacks, companies typically run static analyzers on their source code before releasing a new version. Running static analyzers helps detect, and potentially fix, software vulnerabilities earlier during development, which drastically decreases the cost of potential losses [2]. However, running all available static analyzers is never an option due to the prohibitive cost of running and configuring them on large, real-world codebases. For example, Google's monolithic codebase consists of 2 billion lines of code for all its services, for which Google, despite its vast resources, can only run 1–2 static analyzers (e.g., FindBugs) at a time to detect software vulnerabilities in such a large codebase [3]. The overhead of this process comes not just from running the analyzers, but also from sifting through the generated results, ignoring potential false positives, to fix any vulnerabilities. The more tools to run, the more potential for false positives to go through, which prior research has identified as one of the main reasons for developers abandoning the use of static analyzers in industry [3].

While prior work has thoroughly investigated how developers in industry use static analyzers [4], few reveal how a company may choose a set of static analyzers to run on a project. To pick the most suitable set of analyzers, a company needs to consider several aspects. First, the set of analyzers should prioritize vulnerabilities of paramount importance to the company. Second, the selection criteria should consider how malicious users may exploit potential vulnerabilities. The existence of malicious users rules out the straightforward approach of using a deterministic set of analyzers because it leaves a fixed coverage gap between vulnerability types. Prior work has shown that malicious users are getting aware of traditional decision-making approaches that defence strategists would employ to hinder potential attacks [5]. Even if the coverage gap between analyzers does not immediately expose a vulnerability to malicious users, it may offer hints about where to probe for vulnerabilities. Therefore, simply using a deterministic set of analyzers that performs well on the historical distribution of attacks is not sufficient. A company may overcome these limitations by randomly choosing a set of analyzers to run according to a probability distribution. That way, there is less information about the defence system in effect that may leak to malicious users. This approach is also relevant to companies that opt for building analysis ecosystems such as Google's Tricorder [6] or Microsoft's CloudBuild [7]. The designers of these ecosystems still need to select the analyzers that should be included in their ecosystems based on their coverage, false positive rate, and ease of use. However, how should they best select these static analyzers?

Together with our industry partners at Oracle, we have explored the problem of finding an optimal set of static analyzers to run. The goal of a team at Oracle Labs is to maximize detecting certain types of vulnerabilities by running a set of static analyzers without exceeding a given resource limitation. Since the team needs to commit their code changes frequently, running all available analyzers would result in long analysis time, hindering developers from checking vulnerability reports in time. Moreover, for each code patch, release management teams at Oracle may prioritize fixing all vulnerabilities of a certain type (e.g., buffer overflow). Thus, the selection scheme should reflect the user's need to detect certain types of vulnerabilities.

To address the practical need of the team at Oracle, in this paper, we present Randomized Best Response (**RBR**), a method for finding an optimal randomization over size-bounded sets of static analyzers with adjustable parameters for quantifying the value of a vulnerability type and the cost of running an analyzer. **RBR** assumes that attackers will optimally respond to whichever randomization that the defender chooses. In **RBR**, the software company (i.e., Oracle) plays the role of the defender and malicious users who try to compromise the software play the attacker role. We model the problem of finding an optimal randomization of analyzers as a Stackelberg security game [8], which we express as a Mixed Integer Linear Program (MILP). We then compute an optimal set of analyzers by solving that MILP.

Modelling any scenario game theoretically requires a description of the participants' utilities, which is often manually specified by domain experts. To estimate the utilities in our model, we use the data from the BegBunch dataset [9] to evaluate the performance of 8 static analyzers for C/C++, which serve as the set of potential candidate analyzers to chose from. To realistically estimate the impact and difficulty of exploiting a detected vulnerability, we use the National Vulnerability Database [10]. Our empirical evaluation shows that **RBR** outperforms a number of natural defence strategies. To verify how our approach fares in practice, we discuss a case study on an old version of the codebase A from Oracle and compare the defence strategy suggested by **RBR** with the defence strategy at Oracle when the old version of the codebase was released. We deploy **RBR** with adjusted parameters towards a team at Oracle Labs and verify that **RBR** is able to suggest a defence strategy that provides a wide range of vulnerability coverages while considering resource usage.

II. Background

A. Stackelberg Games

Stackelberg games [11] are non-cooperative games between two players: a leader L and a follower F. Each player i in a Stackelberg game has a set of available *actions* A_i, and the player i decides the action to take according to their *strategy* $s_i \in S_i = \Delta(A_i)$, where $\Delta(X)$ is the set of all distributions over X. We use *action profile* $(a_L, a_F) \in A_L \times A_F$ to denote that the leader performs the action a_L and the follower performs the action a_F. Similarly, we use *strategy profile* $(s_L, s_F) \in S_L \times S_F$ when the leader and the follower have the strategies s_L and s_F, respectively. In particular, a strategy is *mixed* if it assigns positive probabilities to multiple actions.

The outcome of a Stackelberg game is fully determined by the realized action profile. Each player i has a *utility function* $u_i : A_L \times A_F \to \mathbb{R}$ that maps an action profile to a *utility* that measures how much the player prefers the outcome, with each player preferring higher-utility outcomes. Both players aim to maximize their expected utilities, where the expectations are taken over the action distributions induced by the strategy profile. Players seek to maximize their own utility function; the utilities of other players are important to a player only insofar as they help predict other players' actions. We overload the notation of the utility function to represent expected utility when the strategies of L and F are given:

$$u_i(s_L, s_F) = \sum_{(a_L, a_F) \in A_L \times A_F} s_L(a_L) \times s_F(a_F) \times u_i(a_L, a_F). \tag{1}$$

Formally, a Stackelberg game proceeds as follows:

1) The leader chooses a strategy $s_L \in S_L$.
2) The follower observes s_L.
3) The follower chooses a strategy $s_F \in S_F$ based on s_L.
4) An action profile (a_L, a_F) is sampled with $a_L \sim S_L$ and $a_F \sim S_F$.
5) Each player i receives utility $u_i(a_L, a_F)$.

B. Stackelberg Equilibrium

In a Stackelberg game, the utility that a player receives depends on both their own actions and the actions of the other player. We define the *best responses* for player i to strategy s_k as a set of actions:

$$BR_i(s_k) = \{a_i \in A_i \mid u_i(a_i, s_k) \geq u_i(a'_i, s_k) \quad \forall a'_i \in A_i\} \tag{2}$$

where s_k is the strategy of the other player. We say action a for player i is a best response to strategy s_k if and only if $a \in BR_i(s_k)$.

In a Stackelberg game, a strategy profile in which both players are behaving optimally is called a *Stackelberg equilibrium*. There can be multiple such equilibria in a game, especially when the follower is indifferent between multiple actions given the strategy of the leader. In this paper, we focus on a particular refinement of Stackelberg equilibrium called the *strong Stackelberg equilibrium* (SSE), in which the attacker breaks ties in favor of the defender [12].

Definition 1 (Strong Stackelberg Equilibrium)**.** A strategy profile (s^*_L, s^*_F) is a *strong Stackelberg equilibrium* if it satisfies the following:

1) $u_L(s^*_L, BR_F(s^*_L)) \geq u_L(s'_L, BR_F(s'_L)) \quad \forall s'_L \in S_L$
2) $s^*_F(a_F) > 0 \implies a_F \in BR_F(s^*_L) \quad \forall a_F \in A_F$
3) $u_L(s^*_L, s^*_F) \geq u_L(s^*_L, a_F) \quad \forall a_F \in BR_F(s^*_L)$

C. Security Games

Security games are a frequently used framework in which security situations are modelled as Stackelberg games [13]. The two players are the *Defender* (i.e., the leader role), and the *Attacker* (i.e., the follower role). In a security scenario, the attacker observes the defender's behaviour over time before taking actions, allowing the attacker to estimate the defender's strategy before performing an attack. This setup motivates the use of Stackelberg games for modelling these scenarios.

In this paper, we use the *compact security games* framework by [14]. The defender seeks to defend a set of *targets* $T = \{t_1, \ldots, t_n\}$ using a set of *resources* $R = \{r_1, \ldots, r_m\}$. The defender chooses a distribution over resources as defence strategy while the attacker observes the defence strategy and chooses a distribution over targets to attack. Each resource $r \in R$ induces a *coverage vector* $\boldsymbol{p^r} \in [0, 1]^n$, with each element p_t^r representing the probability that target t will be covered by r. When the attacker attacks a covered target, the attack fails; otherwise, the attack succeeds. When an attack on target t succeeds, the attacker receives utility $u_a^1(t)$ and the defender receives utility $u_d^1(t)$. When an attack on target t fails, the attacker receives utility $u_a^0(t)$ and the defender receives utility $u_d^0(t)$. These utilities are not constrained to be zero-sum; i.e., it is not necessarily the case that $u_d^1(t) = -u_a^1(t)$.

As an extension of the original security game model [14], we use a function $cost_i(r,t)$ where $i \in \{a,d\}$ to represent the cost of deploying a resource r or attacking a target t. The utility of a player i when the attacker attacks target t and the defender chooses a resource r with coverage vector $\boldsymbol{p}^r$ is

$$u_i(\boldsymbol{p}^r, t) = p_t^r u_i^0(t) + (1 - p_t^r) u_i^1(t) - \mathrm{cost}_i(r,t). \quad (3)$$

When the defender chooses a mixed strategy $s_d \in \Delta(R)$ and the attacker chooses a mixed strategy $s_a \in \Delta(T)$, the expected utilities are computed in the straightforward way as

$$u_i(s_d, s_a) = \sum_{r \in R} \sum_{t \in T} s_d(r) \times s_a(t) \times u_i(\boldsymbol{p}^r, t). \quad (4)$$

III. Security Games for Vulnerability Detection

To reduce the prohibitive cost of running all available static analyzers, and to avoid running a deterministic set of analyzers, we present **RBR**, a security-game model for finding an optimal randomized set of static analyzers to run. In **RBR**, a software publisher (e.g., Oracle) plays the role of the defender, while a malicious user is an attacker. The attacker seeks to take advantage of a set of vulnerabilities V in the source code of the defender, while the defender aims at preventing exploitation while still providing access to their services. We assume all attackers have the same utility function and, therefore, the optimal strategy for the defender generalizes over all games. The single attacker that we are assuming is in reality an aggregate of many different attacker types. The data that we use to estimate the attacker utility model are for an aggregate typical attacker, so it makes sense that we use a typical attacker approach as well. In reality, different attackers might have different utility functions due to their ability to exploit a vulnerability, but it is straightforward to extend our **RBR** to multiple attacker types by encoding the interaction as a Bayesian game against a distribution of attacker types.

We associate each analyzer with its accuracy and probability of detecting vulnerabilities. **RBR** constrains the number of analyzers that the defender can use by a *budget* b. The set S of resources thus consists of all subsets of analyzers with size b. We refer to these subsets as *schedules* of analyzers with a budget b. The defender chooses a distribution over schedules as a defence strategy before the attacker chooses which vulnerability to attack. **RBR** finds an optimal distribution, which is chosen once, and then sampled from whenever the analyzers are to be run.

A. Utility Model

In our security game model, the company plays the role of the defender by sampling a schedule as the defender strategy according to a fixed probability distribution. Each schedule s has an associated probability $p(s,v)$ of detecting an attack targeting v, and a cost $c_d(s)$ of running the schedule. In our empirical evaluation, we estimate the detection probabilities, runtime, configuration cost, and accuracy of a schedule based on the performance of the underlying analyzers. We then calculate the utility for each schedule based on those estimations. **RBR** may be further extended to include other resource costs.

The attacker, which aims to explore and exploit a vulnerability, chooses a vulnerability type to attack. Each vulnerability type has an impact on the defender when the attack succeeds. We say that a defender receives a negative reward $r_d(v)$ for failing to detect a vulnerability of type v. The attacker receives a reward $r_a(v)$ for successfully attacking v, and incurs a cost $c_a(v)$ for attacking v. We estimate the cost c_a based on the exploitability of the vulnerability. We estimate the reward $r_a(v)$ to the attacker and a negative reward $r_d(v)$ to the defender of a successful attack on v based on the impact of the exploited vulnerability. The attacker's utility from exploiting vulnerability type v when the defender chooses schedule s is

$$u_a(s,v) = (1 - p(s,v)) r_a(v) - \gamma_a c_a(v) \quad (5)$$

where γ_a is the relative weighting of the units of reward and the units of cost. The utility for the defender is

$$u_d(s,v) = (1 - p(s,v)) r_d(v) - \gamma_d c_d(s) \quad (6)$$

where γ_d is the trade-off between units of cost and units of reward. The reward for a detected attack is 0 for the defender.

B. Finding Optimal Randomized Strategies

To solve for an SSE for the game defined above, we encode the game as a MILP. Assuming that the attacker will best respond, solving that MILP efficiently computes the optimal schedule randomization for the defender.

The defender's decision variables are the set $\{\widetilde{p}_s \mid \forall s \in S\}$, where $\widetilde{p}_s$ is the probability of running s. Constraints (10) and (11) ensure that each $\widetilde{p}_s$ is a valid probability. The attacker's decision variables are the set $\{\widetilde{y}_v \mid \forall v \in V\}$, where $\widetilde{y}_v$ is the probability of attacking vulnerability type v. Constraints (8) and (9) ensure that exactly one element of $\widetilde{y}$ is set to 1, with all others set to 0; i.e., the attacker is constrained to deterministically attack a single vulnerability type.

Constraint (12) forces the attacker strategy $\widetilde{y}$ to best respond to the defender's strategy. For the exploited vulnerability type v^*, $\widetilde{U_a}$ must exactly equal the attacker's expected utility $\sum_{s \in S} \widetilde{p}_s u_a(s, v^*)$, because the difference between $\widetilde{U_a}$ and the attacker's expected utility must be both weakly greater and weakly less than 0. For every other unexploited vulnerability type v, the same variable $\widetilde{U_a}$ must be weakly greater than the expected utility of attacking v; i.e., there must not be any other vulnerability types with a strictly greater expected utility, given the strategy of the defender. The constant Z is an arbitrarily large value; its presence allows us to upper bound the difference between $\widetilde{U_a}$ and expected utility for v^* at 0, while removing the upper bound for the other vulnerabilities.

$$\begin{aligned}
\text{maximize} \quad & \widetilde{U_d} - \alpha \sum_{t \in T} \widetilde{c}_t K(t) && (7)\\
\text{subject to} \quad & \widetilde{y}_v \in \{0,1\} \quad \forall v \in V && (8)\\
& \sum_{v \in V} \widetilde{y}_v = 1 && (9)\\
& 0 \le \widetilde{p}_s \le 1 \quad \forall s \in S && (10)\\
& \sum_{s \in S} \widetilde{p}_s = 1 && (11)\\
& 0 \le \widetilde{U_a} - \sum_{s \in S} \widetilde{p}_s u_a(s,v) \le (1-\widetilde{y}_v) Z \quad \forall v \in V && (12)\\
& \widetilde{U_d} - \sum_{s \in S} \widetilde{p}_s u_d(s,v) \le (1-\widetilde{y}_v) Z \quad \forall v \in V && (13)\\
& \widetilde{c}_t \in \{0,1\} \quad \forall t \in T && (14)\\
& \widetilde{p}_s \le \widetilde{c}_t \quad \forall s \in S, t \in T(S) && (15)
\end{aligned}$$

Constraint (13) serves a similar role for the defender's expected utility. $\widetilde{U_d}$ is required to be weakly less than the defender's expected utility given its mixed strategy $\widetilde{p}$ and the attacked vulnerability type. Since $\widetilde{U_d}$ is the maximization objective, there is no need for a lower bound to force equality.

We extend the classical encoding of an SSE by introducing Constraint (14) and Constraint (15). One particularity of analyzer selection is how to model the configuration cost that a company must spend before using an analyzer. Although configuration cost is an attribute of the analyzer itself, incorporating it into $u_d(s,v)$ fails to realistically model practical scenarios. For example, let's assume that **RBR** suggests the strategy: $p(\{A,B\}) = 0.8$, $p(\{A,C\}) = 0.1$, and $p(\{C,D\}) = 0.1$. The solution means that the defender must first deploy 4 analyzers (A, B, C, and D), despite running only two of them at a time. To precisely model the configuration cost in **RBR**, we use T to represent the set of candidate analyzers and overload it as a function that maps a schedule to the set of all analyzers used in this schedule. Thus, Constraint (15) restricts the MILP to only assign a non-zero probability to schedules only if all analyzers in the schedule have been configured. The configuration cost is then subtracted from the defender's utility after being scaled with a user-defined parameter α.

IV. Estimating Parameters of Our Utility Model

Our utility model has a number of free parameters: the set V of vulnerability types, the set R of available analyzers, the set S of schedules, the probabilities $p(s,v)$ of detection, the coefficient α that scales the configuration cost, the costs c_a and c_d, the rewards r_a and r_d, and the tradeoffs γ_a and γ_d between rewards and costs. To use **RBR**, a company is free to use its own criteria to estimate these parameters to reflect their best needs. In this section, we will present our estimates for these parameters based on the needs of our industry partners at Oracle. To achieve that, we use the BegBunch dataset [9] and the National Vulnerability Database (NVD) [10].

TABLE I: CWE types that we consider in our evaluation.

CWE Name	CWE ID	CWE Name	CWE ID
Command Injection	78	Memory Leak	401
Classic Buffer Overflow	120	Use After Free	416
Integer Overflow	190	Use of Uninitialized Variable	457
Unchecked Return Value	252	NULL Pointer Dereference	476

A. Impact and Exploitability

To estimate the severity of a vulnerability in **RBR**, we use the NVD [10] impact and exploitability scores of Common Vulnerabilities and Exposures (CVEs). The higher an impact score is, the more dangerous the exploit is. The higher the exploitability score of a vulnerability is, the easier it is to exploit. Given a CWE type v, we compute $r_a(v)$ as the mean value of impact scores of all CVE instances if the CVE instance is related to the CWE type v. Correspondingly, we set $r_d(v) = -r_a(v)$. However, our final utility model is not zero-sum, because the costs of the attacker and the defender differ. Similarly, we set $c_a(v)$ to be the mean value of the exploitability score for all CVE instances related to the CWE type v, which follows the idea of how MITRE calculates the score of the top CWEs [15] each year.

B. Vulnerability Detection Probability

An important metric in evaluating a static analyzer is how many true positives (i.e., vulnerabilities that are detected successfully) it finds in a given codebase. We use the *recall* value for each CWE vulnerability type v as the detection probability of a static analyzer a. Formally,

$$p(a,v) = \frac{TP}{TP + FN} \tag{16}$$

where FN is the number of vulnerabilities that the analyzer fails to detect. For a schedule s, we say s detects a vulnerability instance b if any analyzer in s recognizes b as a vulnerability. To calculate $p(s,v)$ for a given schedule s and vulnerability type v, we run all analyzers in s on the Accuracy suite from the BegBunch dataset [9] and collect the number of TPs and FNs. We then use Equation (16) as the detection probability for schedule s.

Based on the 8 CWE types that we focus on, we consider 8 C/C++ potential static analyzers that a team at Oracle may use: Parfait [16], Uno [17], Infer [18], Klee [19], Cppcheck [20], Scan-build [21], Flawfinder [22], and Splint [23] as the candidate analyzers to choose a schedule from. The development team will not run all of the analyzers but only a subset of them. Each analyzer may detect some or all CWEs. At first glance, considering 8 analyzers may be a small set of candidates. However, our discussions with the team at Oracle have shown that, in practice, the team would not run more than 2–3 deep static analyzers (e.g., taint trackers). Thus, we evaluate schedules with budgets varying from $b = 1$ to $b = 4$.

C. Defender Costs

The cost of using a static analyzer depends on several factors. Prior work has shown that runtime and false positive rate are of paramount importance to developers [3]. If an analyzer takes too long to terminate or generates too many false alarms, it hinders the developer from properly using the analysis results to locate and fix potential vulnerabilities. Even worse, it makes the developer lose faith in the analyzer and tend to ignore its vulnerability reports. To avoid bias towards analyzers that do not output alarms (i.e., inherently have low false positive rate), the cost of an analyzer $c_d(s)$ depends on the accuracy rate, which we define as:

$$acc = \begin{cases} 1 - \frac{2 \times Precision \times Recall}{Precision + Recall} & \text{if } Precision + Recall \neq 0 \\ 1 & \text{otherwise}, \end{cases} \tag{17}$$

$$Precision = \begin{cases} \frac{TP}{TP+FP} & \text{if } TP + FP \neq 0 \\ 1 & \text{otherwise} \end{cases}$$
$$Recall = \begin{cases} \frac{TP}{TP+FN} & \text{if } TP + FN \neq 0 \\ 1 & \text{otherwise} \end{cases}$$

Accuracy indicates how reliable an analyzer is. The closer the accuracy to 0, the better *precision* (i.e., reported vulnerabilities are true vulnerabilities) and *recall* (i.e., unlikely to miss a true vulnerability) the analysis has. Thus, an analyzer with a lower accuracy has a lower cost. Using accuracy instead of FP rate enables us to measure the performance of an analyzer comprehensively since we are neither favouring analyzers that are too cautious to report any non-superficial vulnerabilities, nor analyzers that are too careless to report all program locations as vulnerabilities.

To evaluate the accuracy of an analyzer on a specific CWE type, we run all 8 analyzers on BegBunch and triage the report related to the desired CWE type into 3 categories:

- **True Positive**: real vulnerability reported by the analyzer.
- **False Positive**: vulnerability reported by the analyzer that does not match a real vulnerability.
- **False Negative**: real vulnerabilities that are not reported by the analyzer.

For a schedule s, a vulnerability v is a True Positive (resp. False Positive) if *any* analyzer in s recognizes v as a True Positive (resp. False Positive); and we say v is a False Negative if *all* of the analyzers in s recognizes v as a False Negative.

An analyzer is generally able to detect more than one type of vulnerability. In our evaluation, we take the mean value of accuracy for all CWE types (Table I) that can be detected by an analyzer as the analyzer's accuracy. However, if the defender needs to further differentiate between CWE types, they can weight the accuracy for detecting vulnerabilities of different CWE types to reflect their detection priorities.

In addition to accuracy, an analyzer cost also depends on the runtime of each analyzer, which we collected during the same evaluation on the BegBunch dataset. Thus, we define the cost of an analyzer as:

$$c_d = e_1 \times \text{runtime} + e_2 \times \text{accuracy} \tag{18}$$

where $e_1 + e_2 = 1$ and $\forall i.e_i \in [0, 1]$. To best reflect the relationship between each component of the cost for an analyzer, we scale all values to the range $[0, 10]$. Specifically, we use a linear function to scale the accuracy and a verified sigmoid function ($\sigma(x) = \frac{20}{1+e^{-x/10}} - 10$) to scale the runtime since there is no upper bound.

D. Configuration Cost

From a project manager's view, configuring a static analyzer into the workflow is a challenging task. For example, practical experience at Oracle indicated that it is almost impossible to deploy some analyzers without adding a vast amount of annotations to the source code.

While it is easy to measure the runtime of an analyzer with system tools, as we did for our evaluation, it is hard to quantify the configuration cost for an analyzer because there is no standard way to configure all analyzers. Thus, we estimate the configuration cost based on the content of the official discussion forums for each analyzer. Specifically, we manually count the proportion of posts related to the analyzer configuration and installation from the forum. The higher that proportion is, the larger the configuration cost.

E. Reward-Cost Tradeoff and Configuration Tradeoff

Since we parameterize **RBR** using several sources of information, there is no clear method of comparing one unit of reward against one unit of cost. For example, the attacker's costs are given by the NVD exploitability score and its rewards are given by NVD impact score, and we have no knowledge of how an attacker may weigh the relative importance of each score. To address this problem, **RBR** defines the parameters γ_a and γ_d in Equations (5) and (6). These parameters represent the trade-off between costs and rewards for the attacker and defender. A relatively low value of γ corresponds to a player who values a unit of reward more than a unit of cost. A high value is for a player who is deeply concerned about costs. These parameters may be tuned to represent different types of players. For example, if $\gamma_a = 0$, then the attacker's utility does not account for costs, analogous to an attacker with more resources than needed for any attack. On the other hand, a large γ_a stands for a cost-cautious attacker.

We have also introduced a coefficient α to represent how a company values the configuration cost. A larger α means the company cannot afford the cost of configuring many analyzers, while a small α means that the company has enough resources to configure more analyzers to potentially find more vulnerabilities. In our evaluation, we fix the value of γ_a, γ_d, and α all equal to 0.1.

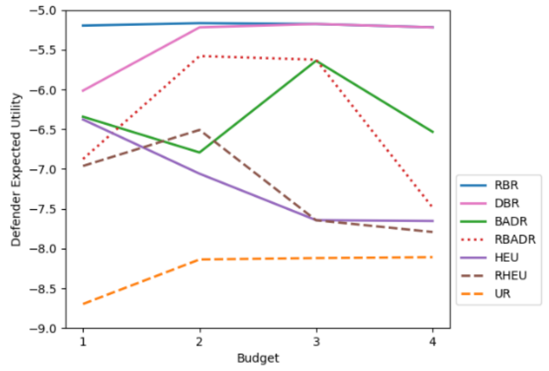

Fig. 1: Comparing **RBR** to various baseline strategies.

V. Evaluation

We evaluate **RBR** through the following research questions:

- **RQ1**: How does **RBR** compare to a set of baseline strategies?
- **RQ2**: How does changing the coefficients for e_1, e_2, and α affect the utility for the defender?
- **RQ3**: How does **RBR** behave in a real-world scenario compared to the current defence strategy used at Oracle?

For **RQ1** and **RQ2**, we ran our experiments on a machine with Intel Core I7-7700HQ 2.8 GHz 4-core processor, 32 GB RAM, running Ubuntu 20.04. For **RQ3**, we used a machine with Intel Xeon E5-2690 2.9 GHz processor, 16 GB RAM, running Oracle Linux 8. To solve the MILP, we used Z3 4.8.15 [24].

A. Comparison to Baseline Strategies (RQ1)

We compare **RBR** against the following baseline strategies in terms of their optimal defender utilities:

- **Uniform Randomization (UR):** the defender uniformly randomizes over all schedules in S.
- **Best Average Detection Rate (BADR):** the defender always runs the schedule s' with the best average detection rate of specific vulnerability types.

$$s' \in \underset{s \in S}{\operatorname{argmax}} \frac{1}{|V|} \sum_{v \in V} p(s \text{ detects } v) \tag{19}$$

- **Randomized Best Average Detection Rate (RBADR):** an extension of **BADR** that uniformly randomizes over the two strategies with highest average detection rates.
- **Highest Expected Utility (HEU):** the defender chooses a schedule s' with the highest expected utility, calculated by taking the expectation of utility over the likelihood exploited vulnerabilities, where the probability of each CWE is taken from the uniform distribution.

$$s' \in \underset{s \in S}{\operatorname{argmax}} \sum_{v \in V} u_d(s, v) \cdot p(v), \tag{20}$$

- **Randomized Highest Expected Utility (RHEU):** an extension of **HEU** that applies uniform randomization over the two strategies with the highest expected utilities.

TABLE II: The schedules and probabilities for **RBR**.

	RBR		
Budget	**Utility**	**Schedule**	**Probability**
1	-5.196	PARFAIT FLAWFINDER	0.792 0.208
2	-5.165	PARFAIT, CPPCHECK FLAWFINDER, CPPCHECK	0.775 0.225
3	-5.176	PARFAIT, CPPCHECK, FLAWFINDER	1
4	-5.218	PARFAIT, FLAWFINDER, UNO, CPPCHECK	1

TABLE III: The schedules for **DBR**.

	DBR	
Budget	**Utility**	**Schedule**
1	-6.014	UNO
2	-5.218	PARFAIT, FLAWFINDER
3	-5.176	PARFAIT, CPPCHECK, FLAWFINDER
4	-5.218	PARFAIT, FLAWFINDER, UNO, CPPCHECK

- **Deterministic Best Response (DBR):** the defender best responds to the attacker by choosing a single schedule s^*. **DBR** is a restricted case of **RBR**, where the defender assigns probability 1 to s^* and 0 to all other schedules.

For all strategies, the defender has a negative expected utility because they cannot win this game; they only seek to minimize their losses. Therefore, the closer to 0 that a strategy's utility is, the better it performs.

Figure 1 shows that, regardless of the budget, **RBR** outperforms all other strategies; except when budget equals 3 and 4, where **RBR** and **DBR** output the same solution. The difference between **RBR** and **DBR** shows the benefit that the defender realizes through randomization. This is because a **DBR** is more easily exploited by the attacker than an **RBR**. For the other strategies, the defender either does not consider the actions of the attacker or resource usage, earning them a lower utility. To further illustrate the differences between **RBR** and **DBR**, Table II and Table III show the solutions suggested by **RBR** and **DBR** along with their expected utilities for various budgets. The tables show that both **RBR** and **DBR** eliminate the use of 4 analyzers: KLEE, INFER, SPLINT, SCAN-BUILD. To better understand the reasons behind this elimination, we examined the analysis runtime, accuracy, and configuration overhead of each analyzer. Table IV shows the measured costs; the smaller the value, the lower the cost of running its corresponding analyzer. The table shows that SCAN-BUILD, KLEE, and INFER have relatively high runtime scores. Moreover, KLEE times out on several benchmarks because it depends on symbolic execution techniques. Additionally, SPLINT is too expensive to configure, because it requires modifying the code to add annotations, which is a tremendous overhead for developers.

TABLE IV: Each cost component of running an analyzer. Smaller values indicate better performance.

Analyzer	Runtime	Accuracy	Configuration
PARFAIT	1.94	2.62	3.80
UNO	0.46	4.13	0.50
SPLINT	0.15	2.81	9.50
SCAN-BUILD	9.04	3.33	3.36
KLEE	9.02	1.41	5.37
INFER	8.73	4.80	8.85
FLAWFINDER	0.58	7.96	0.50
CPPCHECK	1.33	4.41	1.00

RQ1: **RBR** outperforms all other strategies, showing that randomization and considering the actions of the attacker increase the defender's optimal utility.

*B. Sensitivity Analysis (**RQ2**)*

In **RBR**, e_1, e_2, and α decide how much the MILP values runtime, accuracy, and configuration cost of an analyzer. The higher the coefficient is, the more it values the corresponding attribute. To measure the effect of varying e_1, e_2, and α on computing an optimal defender utility, we increase the value of each coefficient separately while keeping the other two parameters equal for budgets 1–4.

Figure 2 shows the change of the optimal defender utility with e_1, e_2 under different budgets while maintaining $e1 + e2 = 1$. Since we are solving an MILP, the solution stays the same unless there is a change in the slope of the curve in each plot in Figure 2. The figure shows that changing any coefficient does not change the value of the optimal defender utility drastically. For budgets 1–4, the defender expected utility has a maximum increase of 0.3 and stays in the range $[-5.35, -5]$ when we increase e_1 from 0 to 1 with a stride of 0.1. Thus, the solution suggested by **RBR** is insensitive to the value of e_1 and e_2 since the set of the choice analyzers remains the same.

On the other hand, varying α has an effect on the defender's expected utility. Figure 3 shows that varying α from 0 to 1 decreases the utility by a maximum value of -6 while also changing the chosen set of analyzers. The solution changes 2–3 times as we increase α from 0 to 1. If we increase the value of α, **RBR** is more likely to assign a higher probability to analyzers that have smaller configuration cost. The results verify that α controls the tradeoff between the vulnerability coverage and the cost of configuring more analyzers efficiently.

RQ2: **RBR** is sensitive to α but insensitive to e_1 and e_2. Increasing α makes **RBR** bias to analyzers that are easy to configure.

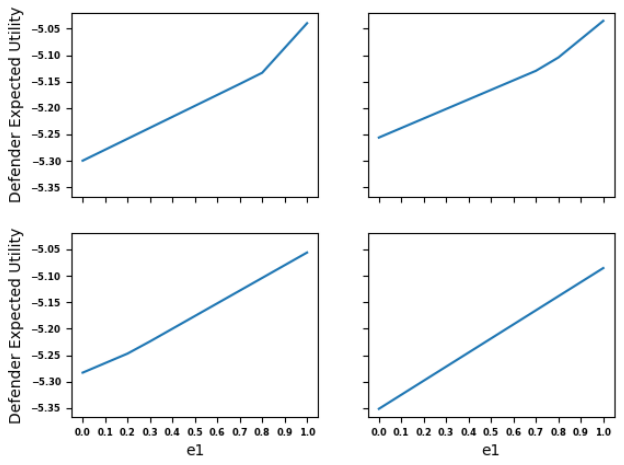

Fig. 2: The change in the optimal defender utility with varying e_1 and e_2. For all runs, $\gamma_a = \gamma_d = 0.1$.

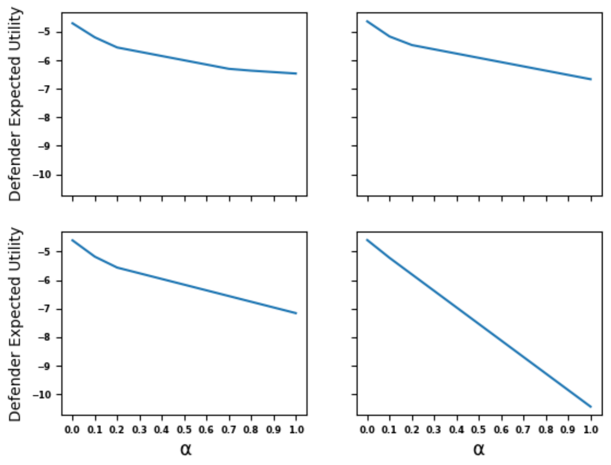

Fig. 3: The change in the optimal defender utility with varying α. For all runs, $\gamma_a = \gamma_d = 0.1$.

*C. Real-World Scenario within Oracle (**RQ3**)*

To show that **RBR** provides useful guidance on choosing analyzers for real-world programs, we use its solution to detect vulnerabilities in one of the codebases (A) at Oracle, which has over one million lines of `C` code. To compare against the vulnerability reports that were manually verified by the Oracle development team, we use an older version of A that was in use during the time when these reports were generated. These vulnerability reports are collected based on an in-house analyzer developed by Oracle. In other words, the defence strategy of Oracle is running the in-house analyzer with probability 1, similar to **DBR** of budget 1. However, these vulnerability reports are not necessarily the full set of all vulnerabilities in the source code. As we will show later, running other analyzers lead to discovering more true positives. For this case study, the objective of the next code patch is to fix vulnerabilities of types that Table I lists. Thus, if the analyzers suggested by **RBR** detect more vulnerabilities of these 8 types than the baseline strategy, then **RBR** provides a better selection strategy.

From Table II, we choose our defence strategy when the budget = 2, because that budget yields the highest defence utility. The defence strategy is a probability distribution over the power set of analyzers. In other words, we will randomly choose a group of analyzers among (FLAWFINDER + CPPCHECK) and (PARFAIT + CPPCHECK) with their associated probabilities as the final solution. However, we show the results for all schedules to avoid favouring any specific one.

To compare the number of vulnerabilities detected by our defence strategy to the baseline strategy, we ran all analyzers in our defence strategy and manually verify if the vulnerability reports are true positives. Table V shows the number of true positives of each schedule along with the number of vulnerabilities found by running the baseline strategy. The last column in the table represents the number of vulnerability reports that we gathered by running the baseline strategy. These are true positives found by the defence strategy and confirmed by the development team at Oracle. Thus, we could verify the performance of the **RBR** defence strategy by comparing the average number of true positives between the **RBR** defence strategy with the baseline strategy.

Across all CWEs, the schedule (PARFAIT, CPPCHECK) finds the highest total number of true positives (282) than the other schedule (FLAWFINDER, CPPCHECK), which detects 238 true positives; both are still more than the baseline strategy (210). Since both schedules detect more true positives, randomly choosing one of them also leads to detecting more vulnerabilities. For each CWE type, each schedule detects a different set of true positives. For example, schedule (FLAWFINDER, CPPCHECK) finds fewer true positives than the baseline for CWE-120, CWE-401, CWE-416, and CWE-457. After manually triaging the true positives, we have confirmed that the main reason is both analyzers in that schedule perform poorly for inter-procedural issues. For CWE-476, all schedules detect more true positives than the baseline due to using CPPCHECK. In fact, CPPCHECK singlehandedly finds all 208 true positives. Thus, the schedule (PARFAIT, CPPCHECK) leads to the largest overall number of detected true positives.

Since Oracle would only run one schedule at a time from the computed **RBR** stategy, we have also calculated the weighted average for each CWE type. The penultimate column in Table V shows the number of true positives that will be detected in the long run when Oracle follow the **RBR** strategy. Overall, **RBR** outperforms the baseline strategy by detecting more true positives. Thus, our **RBR** strategy can be used in real word programs to select the best set of analyzers.

RQ3: Our experience of using **RBR** at Oracle shows that **RBR** suggests a set of analyzers that detect more true positives than the existing strategy that Oracle has, while considering resource usage and the attacker choices.

VI. LIMITATIONS

Our approach to estimating the parameters for **RBR** has a few limitations.

a) Limited bug instances for a CWE type: Although BegBunch contains over 2,300 programs, most of the benchmarks only contain bugs of certain types. Thus, for some CWE types that we considered in our evaluation, fewer than 20 bug instances are used in estimating the precision and recall of an analyzer. This is a potential cause of inaccurate estimation.

b) Limited CVE availability: A similar issue also appears in the NVD dataset. For certain types of CWE bugs in our evaluation, the NVD dataset for the most recent 10 years may only contains fewer than 10 bug instances exploited and reported in existing software.

c) Classifying a CWE type: Not all analyzers output a detailed, user-friendly results. Most would only output a description and a location for each found bug. For these analyzers, we have to manually collate the output bug information to a CWE bug type. However, it is sometimes hard to exactly map a description to the CWE type as some analyzers may output the same description for bugs of different CWE types. But only about 5% of bug reports fall into this category, thus we believe that this issue would only slightly degrade the quality of estimation.

VII. RELATED WORK

A. Security Games

Over the last decade, Stackelberg security games have been applied to a wide variety of security domains. [25] provide an excellent overview of the field. In this section, we highlight two particularly relevant works.

Our work is inspired by the work of [8], which employs the use of Stackelberg security games to assign security resources in the LAX airport. Their work innovates by considering the different values for defense targets. While our work incorporates many ideas from their paper, we also present several improvements to their model. The first, and most important difference is that they apply their model for security in a traditional, physical setting, whereas our approach is for a more abstract defense. Attacks in the domain of bug detection are much lower risk for an attacker to make compared to terror attacks carried out in-person. Therefore, our domain, naturally, has a higher ratio of attackers to defenders. Additionally, while their model considers the several factors when judging the value of attacking certain targets, it does not consider the difficulty of carrying out an attack on the target, which our model incorporates.

To model the assignment of transit police patrol scheduling, [26] applied security problems on the domain of graph

TABLE V: True positives for **RBR** strategy. **RBR** suggests using one of the schedules with the associated probabilities. Each line shows the number of true positives detected by the possibly used schedule. The last column shows the number of true positives detected by the baseline strategy at Oracle.

	Schedule of RBR Strategy (Probability)			**Baseline Strategy**
Vulnerability	PARFAIT + CPPCHECK (0.775)	FLAWFINDER + CPPCHECK (0.225)	Weighted Average	
CWE-78	**0**	**0**	**0**	0
CWE-120	**22**	16	**20**	18
CWE-190	**0**	**0**	**0**	0
CWE-252	**0**	**0**	**0**	0
CWE-401	**5**	1	4	5
CWE-416	**1**	0	**1**	1
CWE-457	**46**	13	**38**	32
CWE-476	**208**	**208**	**208**	154
Sum	**282**	**238**	**272**	210

patrolling. Like our own work, but unlike the work of [8] which considers terrorist attacks, [26] model a domain in which attacks are relatively low-cost, and thus may frequently succeed. Since attackers are commuters with routines, they plausibly will only change their decision of whether or not to buy a ticket, whereas attackers in our model may always change the type of their attack.

B. Game Theory for Cybersecurity

Game theory and security games have historically been used to optimize cybersecurity defenses. In our discussion, we will focus more on the application of game theory to digital defenses than how to more effectively detect new malware.

Zaffarano et al. [27] present a system for designing metrics for modelling non-static software defense. Our work expands on their model by adding costs for attackers and defenders, making the game non-zero sum. This change allows our model to avoid computing sub-optimal strategies as a result of ignoring these costs.

Chung et al. [28] criticize the use of game theoretic approaches to malware detection, including vulnerability to novel attacks, and a lack of a true measure of costs and rewards. The authors present a game theoretic approach for automated responses to network breaches, allowing systems to defend themselves without the need for human administrator action. Their approach specializes in domains where knowledge of attacks and their payoffs is limited, which is not the case in our work.

Wang et al. [29] present a many-player game theoretic approach to security in mobile ad-hoc networks (MANETs). Their model allows normal users in MANETs to make distributed security defense decisions, allowing the MANET to be more resilient to being compromised, while also minimizing the resources required to do so. Their work uses system resource constraints as a budget, which is similar to our use of budgets as a constraining factor.

Game theory has also been employed in "honeypotting," where vulnerable "bait" systems are distributed to hide real ones, to waste attacker time, warn against incoming attacks, and understand new vulnerability exploits. [30] uses game theory to optimally decide how vulnerable to make the bait systems, as well as how many to use, to maximize their effectiveness. Their later work [31] considers attackers who use probes to detect if a system is real, and uses attack graphs to better model an attacker's knowledge of a system. Attack probes explain why attackers can be best responding (by having perfect information) in our model, as an attacker is able to repeatedly probe our defenses to see which attacks are more likely to succeed.

VIII. CONCLUSION

The interaction between a company and malicious users may be viewed as a game. The company aims to publish its software in the least exploitable way while malicious users try to do as much damage as possible by exploiting vulnerabilities in the software. To detect and eventually fix the vulnerabilities in the source code, the company needs to run static program analyzers before releasing their code. There are plenty of static program analyzers for different languages and each with a different vulnerability coverage. Even though a company may have a large infrastructure that enables them to run a large set of analyzers that cover most vulnerabilities, it is impossible to provide full coverage due to the prohibitive cost of deploying a static analyzer. An immediate question is how to choose the set of analyzers to run. Using a deterministic subset of available analyzers would also leave a deterministic gap between vulnerability types. To prevent the attacker from estimating the chosen set of analyzers to find vulnerabilities in the source code, the company needs to randomly choose the set of analyzers to run.

In our work, we have presented **RBR**, a method for computing an optimal randomization over available static program analyzers by modelling the interaction between the software publisher and malicious users who try to exploit vulnerabilities in the software. In **RBR**, the defender's costs depend on a few objective factors (e.g., accuracy for different tools on different attacks) and subjective factors (e.g., relative costs of different vulnerabilities being exploited). We classify vulnerabilities according to their CWE categories and estimate the objective factors from the National Vulnerability Dataset [10]. However, the **RBR** parameters can and should be customized for different users because there could be a disparity in how organizations define the value of a vulnerability. Thus, users should choose the parameters that fit their needs best before using **RBR**. Our empirical evaluation shows that **RBR** outperforms all other baseline strategies with respect to the defender utility. We also conduct a sensitivity analysis to show how varying the user-defined coefficient will affect the model output. Finally, we have performed a case study of our method with one of the large systems with more than a million lines of code from Oracle, which shows how **RBR** may be deployed in a real-world scenario.

REFERENCES

[1] H. Krasner, "The cost of poor software quality in the us: A 2020 report," 2021. [Online]. Available: https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf

[2] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," 2002.

[3] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. ACM, 2016, pp. 332–343. [Online]. Available: https://doi.org/10.1145/2970276.2970347

[4] N. Chong, B. Cook, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Code-level model checking in the software development workflow," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 11–20. [Online]. Available: https://doi.org/10.1145/3377813.3381347

[5] R. S. Gutzwiller, K. J. Ferguson-Walter, and S. J. Fugate, "Are cyber attackers thinking fast and slow? exploratory analysis reveals evidence of decision-making biases in red teamers," *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 63, no. 1, pp. 427–431, 2019. [Online]. Available: https://doi.org/10.1177/1071181319631096

[6] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 598–608. [Online]. Available: https://doi.org/10.1109/ICSE.2015.76

[7] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "Cloudbuild: Microsoft's distributed and caching build service," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 11–20. [Online]. Available: https://doi.org/10.1145/2889160.2889222

[8] M. Jain, J. Tsai, J. Pita, C. Kiekintveld, S. Rathi, M. Tambe, and F. Ordóñez, "Software assistants for randomized patrol planning for the LAX airport police and the federal air marshal service," *Interfaces*, vol. 40, no. 4, pp. 267–290, 2010. [Online]. Available: https://doi.org/10.1287/inte.1100.0505

[9] NIST, "The software assurance metrics and tool evaluation (samate) project," 2022.

[10] H. Booth, D. Rike, and G. Witte, "The national vulnerability database (nvd): Overview," 2013-12-18 2013. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=915172

[11] H. von Stackelberg, D. Bazin, R. Hill, and L. Urch, *Market Structure and Equilibrium*. Springer Berlin Heidelberg, 2010. [Online]. Available: https://books.google.ca/books?id=dghH9OH5fDoC

[12] B. Von Stengel and S. Zamir, "Leadership with commitment to mixed strategies," Citeseer, Tech. Rep., 2004.

[13] R. Avenhaus, B. von Stengel, and S. Zamir, "Chapter 51 inspection games," *Handbook of Game Theory with Economic Applications*, vol. 3, pp. 1947–1987, 12 2002.

[14] C. Kiekintveld, M. Jain, J. Tsai, J. Pita, F. Ordóñez, and M. Tambe, "Computing optimal randomized resource allocations for massive security games," in *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 1*. IFAAMAS, 2009, pp. 689–696. [Online]. Available: https://dl.acm.org/citation.cfm?id=1558108

[15] CWE, 2022. [Online]. Available: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

[16] C. Cifuentes and B. Scholz, "Parfait - designing a scalable bug checker," in *Scalable Program Analysis, 13.04. - 18.04.2008*, ser. Dagstuhl Seminar Proceedings, vol. 08161. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2008. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2008/1573/

[17] G. Holzmann and M. Hill, "Uno: Static source code checking for user-defined properties 1," 2002.

[18] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of C programs," in *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, vol. 6617. Springer, 2011, pp. 459–465. [Online]. Available: https://doi.org/10.1007/978-3-642-20398-5_33

[19] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[20] CppCheck, 2022. [Online]. Available: https://cppcheck.sourceforge.io/

[21] Scan-build, 2022. [Online]. Available: https://clang-analyzer.llvm.org/scan-build.html

[22] FlawFinder, 2022. [Online]. Available: https://dwheeler.com/flawfinder/

[23] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Softw.*, vol. 19, no. 1, pp. 42–51, 2002. [Online]. Available: https://doi.org/10.1109/52.976940

[24] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[25] A. Sinha, F. Fang, B. An, C. Kiekintveld, and M. Tambe, "Stackelberg security games: Looking beyond a decade of success," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. ijcai.org, 2018, pp. 5494–5501. [Online]. Available: https://doi.org/10.24963/ijcai.2018/775

[26] Z. Yin, A. X. Jiang, M. Tambe, C. Kiekintveld, K. Leyton-Brown, T. Sandholm, and J. P. Sullivan, "TRUSTS: scheduling randomized patrols for fare inspection in transit systems using game theory," *AI Mag.*, vol. 33, no. 4, pp. 59–72, 2012. [Online]. Available: https://doi.org/10.1609/aimag.v33i4.2432

[27] K. Zaffarano, J. Taylor, and S. Hamilton, "A quantitative framework for moving target defense effectiveness evaluation," in *Proceedings of the Second ACM Workshop on Moving Target Defense, MTD 2015, Denver, Colorado, USA, October 12, 2015*. ACM, 2015, pp. 3–10. [Online]. Available: https://doi.org/10.1145/2808475.2808476

[28] K. Chung, C. A. Kamhoua, K. A. Kwiat, Z. T. Kalbarczyk, and R. K. Iyer, "Game theory with learning for cyber security monitoring," in *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, 2016, pp. 1–8.

[29] Y. Wang, F. R. Yu, H. Tang, and M. Huang, "A mean field game theoretic approach for security enhancements in mobile ad hoc networks," *IEEE Transactions on Wireless Communications*, vol. 13, no. 3, pp. 1616–1627, 2014.

[30] R. Píbil, V. Lisý, C. Kiekintveld, B. Bosanský, and M. Pechoucek, "Game theoretic model of strategic honeypot selection in computer networks," in *Decision and Game Theory for Security - Third International Conference, GameSec 2012, Budapest, Hungary, November 5-6, 2012. Proceedings*, ser. Lecture Notes in Computer

Science, vol. 7638. Springer, 2012, pp. 201–220. [Online]. Available: https://doi.org/10.1007/978-3-642-34266-0_12

[31] C. Kiekintveld, V. Lisý, and R. Píbil, “Game-theoretic foundations for the strategic use of honeypots in network security,” in *Cyber Warfare - Building the Scientific Foundation*, ser. Advances in Information Security. Springer, 2015, vol. 56, pp. 81–101. [Online]. Available: https://doi.org/10.1007/978-3-319-14039-1_5