OpenSAW: Open Security Analysis Workbench

Noomene Ben Henda[1], Björn Johansson[1], Patrik Lantz[1], Karl Norrman[1(✉)], Pasi Saarinen[1], and Oskar Segersvärd[2]

[1] Ericsson Research Security, Stockholm, Sweden
{noamen.ben.henda,bjorn.a.johansson,patrik.lantz, karl.norrman,pasi.saarinen}@ericsson.com
[2] School of CSC, Royal Institute of Technology (KTH), Stockholm, Sweden
oskarseg@kth.se

Abstract. Software is today often composed of many sourced components, which potentially contain security vulnerabilities, and therefore require testing before being integrated. Tools for automated test case generation, for example, based on white-box fuzzing, are beneficial for this testing task. Such tools generally explore limitations of the specific underlying techniques for solving problems related to, for example, constraint solving, symbolic execution, search heuristics and execution trace extraction. In this article we describe the design of OpenSAW, a more flexible general-purpose white-box fuzzing framework intended to encourage research on new techniques identifying security problems. In addition, we have formalized two unaddressed technical aspects and devised new algorithms for these. The first relates to generalizing and combining different program exploration strategies, and the second relates to prioritizing execution traces. We have evaluated OpenSAW using both in-house and external programs and identified several bugs.

1 Introduction

Background. Dynamic test generation is a testing technique where the test inputs are automatically generated while running the System Under Test (SUT) in a continuous loop. In each iteration, the new test input is generated based on information collected during the execution of the SUT on previously generated input. When the SUT is an executable binary program, that for example reads an input file, the technique consists in performing the following procedure. First, the program is executed on an initial file obtaining a trace of executed instructions; second, the trace is used to generate a new input file; and finally, the previous two steps are iteratively applied using the new file as input. Such a procedure can continue running until a termination criterion is reached, for example a timeout or a user interruption. The process of input generation usually relies on symbolic execution and constraint solving, which we briefly explain as follows. A program trace consists of instructions executed by the processor. Among these instructions are conditional ones such as "jump if equal". Conditional instructions have two possible outcomes or branches. For each conditional instruction that occurs in a trace, one of the branches must have been taken. Given a trace

© Springer-Verlag GmbH Germany 2017
M. Huisman and J. Rubin (Eds.): FASE 2017, LNCS 10202, pp. 321–337, 2017.
DOI: 10.1007/978-3-662-54494-5_18

and a conditional instruction, we are interested in generating a new input that causes the program to take the other untaken branch of the instruction. For that, symbolic execution help us generate the constraint on the input that could potentially achieve this. Symbolic execution maps the input bytes to symbolic variables and then emulates the execution of the trace instruction while tracking these variables. During the simulation, the symbolic variables are assigned expressions reflecting the effect of the instructions. When a conditional instruction is reached, a constraint is generated by substituting, in the condition, the variables in scope by their corresponding expressions. Within a trace, the conjunction of all the branch instruction constraints defines what often is referred to as the *path condition*. For our goal, we only need to stop at the branch instruction of interest The constraint is then fed to a constraint solver, which delivers the desired input.

In academia the technique is sometimes referred to as white-box fuzzing [12]. Compared to black-box fuzzing, used by, for example, AFL [23] and Sulley [22] where the program input is randomly generated, the technique is guaranteed to achieve better code coverage and thus is more likely to find bugs. Think of a program that first tests if the input is equal to a certain value and just exits otherwise. With black-box fuzzing it is more likely that we never pass the test but with this technique we will be able to generate an input that passes the test after the first iteration. The drawback is that white-box fuzzing is slower. Nevertheless the technique will always benefit from the continuous increase in efficiency and speed of SAT solving [2] which is the underlying key procedure in constraint solving and symbolic execution. Currently, there are several academic [5,8] and industrial [11] frameworks implementing the technique and efficiently used to test industrial applications. In particular, for security testing the technique is used to detect vulnerabilities by generating input that can cause the program to crash, or that can allow for hijacking the execution, etc [7]. Other use-cases involve back-door detection and malware analysis for detecting unwanted functionality and behavior in binaries [13].

One of the challenges that we identified in this approach is the following. Given an execution trace, how to best select the branch instruction, i.e., the conditional instruction for which to trigger the other branch. The problem is that there is no well-defined generic technique for such selection strategies. Existing frameworks differ greatly in their implementations. A systematic approach that selects each branch in each encountered trace would work for small programs with small input domain. In such cases the technique could potentially cover all possible execution paths in a reasonable time. For real size applications like compilers, or document processors such an approach is inefficient. One would think of a strategy that spreads the search across the execution path and avoid exploring already covered portions of code. This would require some sort of code book-keeping. In some other cases one would possibly prefer a more focused selection strategy (in a Depth-first search manner) for traces containing portions of code for example from newly added or upgraded libraries.

One other challenge is the following. In parallel settings where several execution traces can be obtained simultaneously, how to best select the trace to consider first. One good measure for selection relies on how many new blocks are visited by the trace. In other terms, one can measure the amount of new program code in the trace compared to the previous traces processed earlier and use that as a ranking measure. There is no clear method on how to effectively implement such method. Obviously, one wants to avoid comparing each new trace with each of the old traces as this is extremely inefficient.

Related Work. In [16] a technique is described for keeping track of execution traces. The technique is based on recording an execution history for each trace. An execution history records all executed instructions and their occurrences. This is a standard data structure used for program slicing techniques. The choice of this structure is for handling symbolic execution rather than for branch (or path) selection strategy. For branch selection, [16] relies on a critical path oriented strategy which is mainly based on choosing instructions that follow the input data (tainted) propagation.

In [12] a technique called Generational search is proposed for branch selection. The technique is based on classifying the generated program inputs according to their generation level. The initial input has level 0. The inputs generated during the ith iteration of the framework have level $i+1$. Given an execution trace obtained by running the program on an i level input, the branch conditions that are selected are the ones that appear in i^{th} position and upwards in the trace. MergePoint [3] is a system which utilizes this strategy by initiating the symbolic execution using a concrete input seed and explore paths using generational search. It adds an additional step in the symbolic execution denoted as path merging in order to reduce the number of explored paths.

Haller et al. introduce yet another search heuristic, denoted as Value Coverage Search (VCS) [13,14]. This strategy identifies potentially vulnerable code regions using static analysis, and then steer the symbolic execution along branches that are probable to lead to those regions. Code regions are deemed potentially vulnerable if they change pointer values. The authors argue that their strategy is superior to other traditional strategies, such as Depth-first search which can also be infeasible to use when the symbolic input size increases. In [14], the strategy is based on weights from a learning phase which are used to steer its symbolic execution toward new and interesting pointer dereferences. During the learning phase each branch is assigned a weight approximating the probability that the path following the direction contains new pointer dereferences.

In Driller [21], a guided fuzzer is described for vulnerability detection. Whenever the fuzzer is stuck and is unable to find new paths through a program, a concolic execution engine is invoked. The engine uses the traces from the fuzzing to identify new input that diverge into new code. When encountering conditional branches, the tool checks if negating the condition would result in execution of undiscovered code. A similar approach is also used in [10], named directed search. There is no mentioning of how to select branch instructions given a trace containing several instructions.

The technique closest to ours is described in [6] where a server is used in order to decide which branch instruction to select. The server uses a heuristic to pick the best instruction. One of the heuristic is based on picking the instruction that were executed the fewest number of times. It is also mentioned that the server can be configured for different heuristics. However there is no description on how this is implemented. A project which both builds and refines upon [6], is the symbolic execution tool KLEE [5]. When encountering branch conditions, KLEE forks a process for each possible path. Each process executes a single instruction within its context. For the case where there are multiple concurrent processes at the same instruction step, a process scheduling algorithm is used to decide which one to execute next.

S2E [8], a platform for selective symbolic execution, includes basic selection strategy such as random, Depth- and Breadth-first strategies and mentions that there is support for other strategies. However, these additional ones are not described. Other projects which support Depth- and/or Breadth-first strategies include [15,19]. In [7,18,20] the focus is on performance of symbolic execution and applications for discovering vulnerabilities rather than selection strategies.

Compared to the other methods, ours is agnostic to the other steps of the testing framework, i.e., how the new inputs are generated or how the program is run. When it comes to trace prioritization, most of the related work does not address the issue, with some exceptions. In [6,12], a code-coverage-block method is used for ranking the traces. In particular, in [12], this measure is used for giving scores to the generated program inputs. The rank of a trace (or the corresponding input) is based on the number of unexplored code blocks that appear in the trace compared to all other encountered traces. In [5] the process scheduling depends on interleaving two strategies, randomly selecting paths at branch conditions and selecting processes which are most likely to reach new code. The latter apply a combination of the minimum distance to uncovered instructions, and whether the process has recently discovered new code.

Contribution. We present OpenSAW, an open, flexible and scalable framework for dynamic test generation. The framework leverages already available tools like constraint solvers and trace extractors, and implements two new methods to solve the issue of branch selection and trace prioritization. More precisely, the framework provides a method that allows customization of the strategies in a flexible manner. The method keeps track of the generated execution traces by storing them in a special directed graph structure, *the trace graph*. In this graph, each node represents a portion of the program code ending in a (possibly conditional) jump instruction. Each edge represents a trace and the relative position of the target node in the trace. We generalize branch selection strategies to functions over trace graphs. Furthermore, the framework delivers a method for ranking of the traces, that is fully integrated with the trace graph updating procedure. The resulting rank value can be used to prioritize the execution traces for parallelization purposes.

Outline. The next section gives an overview of the framework. Section 3 describes in details the trace graph concept. Section 4 lists some of the features of OpenSAW. Section 5 presents some of our experiments with the framework. Finally, Sect. 6 concludes the article with some future work.

2 Overview of the Framework

2.1 Preliminaries

For a sequence s, we use $|s|$ to denote its length, and $s[i]$ to denote its ith element for all $1 \leq i \leq |s|$. We let $s[i]$ be $\perp$ for all $i > |s|$. Given two sequences s, s', we use $s \sqsubseteq s'$ to denote that s is a subsequence of s'. A program input is a sequence of bytes and a trace is a sequence of binary (or assembly) instructions corresponding to a run of the program on a particular input. We will be only working with finite inputs, and we assume that programs are deterministic and that they do always terminate. We let i and t range respectively over program inputs and traces. Given a program p, for an input i, we denote by $tr(p, i)$ the trace obtained by running p on i. We use $\mathcal{I}$ and $\mathcal{T}$ to denote respectively the set of all possible inputs and traces. For a trace t, we denote by $in(p, t) := \{i \in \mathcal{I} | tr(p, i) = t\}$, i.e. the set of all inputs on which p generates t. Sometimes we omit p, and write $tr(i)$ or $in(t)$ whenever the program is clear from the context.

In general a framework such as ours generates new inputs based on the conditional branch instructions in the program traces. For that, we adopt a less granular definition for traces and let a trace be a sequence of *instruction blocks* or *blocks* for short. Given a trace t, a block is a maximum length subsequence of consecutive instructions containing no branch instructions except maybe in the last position. Observe that this definition is similar to basic blocks in the context of Control Flow Graphs (CFG). The main difference being that blocks may have multiple entry points. As we are dealing with individual traces, it is not possible to know in advance how many entry points a block has. Following our definition, in a trace t, any conditional branch instruction will be at the end of a unique block from t. We will use $\mathcal{B}$ to denote the set of all possible blocks.

Let's fix a program p. Given an input i, the corresponding trace $t = tr(i)$ and a block of interest $b \in t$, the framework performs a series of operations in order to generate a new input that can potentially trigger the other branch of b. More precisely, the framework first performs symbolic execution on t up-to the instruction of interest in order to generate a *path condition*. A path condition is a conjunction of formulas on symbols corresponding to the bytes of $in(t)$. Each such formula represents the condition on the taken branch of one of the encountered conditional instructions in t up-to and including the one in b. Second, the framework generates a new path condition by negating the formula corresponding to the branch of interest. Afterwards, the framework queries a constraint solver for a possible solution. In case a solution is found, the framework uses it in order to finally construct a new input based on the previous input i. We will hide and collapse all theses steps into a single function $gen(t, b)$ whose co-domain is in $\mathcal{I} \cup \{\perp\}$.

2.2 Architecture

During the development of OpenSAW, the main focus was on flexibility and performance. For flexibility, the goal is that the framework should be as agnostic as possible to the underlying used tools such as for symbolic execution, constraint solving. For performance, the aim is that it should be able to distribute and parallelize the tasks so that it benefits as much as possible from the available computational resources. As a result, the framework architecture is as illustrated in Fig. 1 and its main procedure in Algorithm 1.

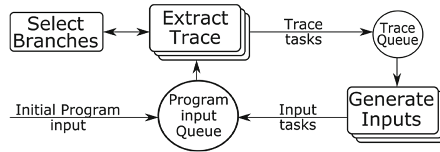

Fig. 1. OpenSAW's architecture

The framework relies on two procedures that can be run in parallel, namely the extractTrace and the generateInputs procedures defined in Algorithm 1. The modules feed each other with tasks through two different priority queues: an input queue and a *trace task* queue denoted respectively by $inpQ$ and trQ in Algorithm 1 (l.1). A trace task is a tuple (t, b) where t is a trace and b is a sequence of blocks from t.

The extractTrace procedure (l.5–13) reads program inputs from an input queue. For each read input i, the procedure first generates the corresponding trace $tr(i)$. From the resulting trace, it then computes a priority $r \in \mathbb{N}$ and selects a number of blocks arranged in a sequence b. Finally, the resulting trace task (t, b) is pushed onto an output queue with priority r. The generateInputs procedure (l.14–25) reads trace task from an input queue. For each obtained task of the form (t, b), the procedures loops through each element in the sequence $b[j]$ for $1 \leq j \leq |b|$ and attempts to generate an input i triggering the untaken branch, i.e., $i' = gen(t, b[j])$. The set iDB is used to keep track of produced inputs that are discarded whenever they are generated again. In case the input i' is both valid and new, it is pushed onto an output queue. For now, we assume that the returned values of the getPriority (l.9) and selectBranches (l.10) functions are computed by an oracle.

For each task that is read, whether it is an input or a trace task, the processing steps can be in fact delegated to worker subprocesses. and hence the nesting representation in Fig. 1. Observe, that the priorities can be rendered useless in case there is no queue buildup. Nevertheless, one can think of a scheme where the priority computation is activated or deactivated depending on the queue size. This and similar implementation details are further discussed in Sect. 4.

Algorithm 1. OpenSAW's main algorithm

Input: A program p and an initial input i_0

```
 1: (iDB, inpQ, trQ) ← (∅, ∅, ∅)
 2: push i0 in inpQ
 3: extractTrace(p, inpQ, trQ)                    ▷ Lines 3–4 are executed in parallel
 4: generateInputs(iDB, trQ, inpQ)
 5: procedure extractTrace(p: program, inQ, outQ: queue)
 6:     while true do
 7:         pop inQ in i                                                    ▷ blocking
 8:         t ← tr(p, i)
 9:         r ← getPriority(t)
10:         b ← selectBranches(t)
11:         push (t, b) in outQ with priority r
12:     end while
13: end procedure
14: procedure generateInputs(iDB: set, inQ, outQ: queue)
15:     while true do
16:         pop inQ in (t, b)                                               ▷ blocking
17:         for j = 1 . . . |b| do
18:             i ← gen(t, b[j])
19:             if i ≠⊥ ∧ i ∉ iDB then
20:                 add i to iDB
21:                 push i in outQ
22:             end if
23:         end for
24:     end while
25: end procedure
```

3 Trace Graphs

3.1 Selection Strategies

One of the central features in such frameworks is the branch selection operation performed, in our case, by the selectBranches function in Algorithm 1. The selection can be done randomly, or by a heuristic [6,12,14], or based on a graph search algorithm such as Breadth-first or Depth-first search. Furthermore, some kind of book-keeping must be performed, for example, in order to avoid repeatedly processing the branch instructions from the same trace. Overall, a strategy is needed. Obviously, the choice of the strategy has a direct impact on the performance of the framework.

In order to define strategies in general, two aspects are taken into consideration. On the one hand, a strategy itself should be easy to change and hence our choice to refactor it out and delegate to an abstract function with a specific interface (selectBranches) unlike for example how it is handled in the SAGE framework. On the other hand, our framework must provide support for a class of strategies as large as possible or at least that subsumes all the ones used in similar frameworks including for example the generational search strategy used

in SAGE. Observe that it is difficult to know what the most general and common prerequisites of a selection strategy are. Nevertheless, the current *history* during a run of the framework provides a good basis for a generic selection strategy.

3.2 Run Data

By history, we mean all the data generated during the run such as the program inputs, the computed priorities, the selected branches but most importantly the program traces. In fact, even for small applications, the number and length of the encountered traces may quickly grow into an unmanageable number. Therefore a new structure is needed to efficiently represent the set of generated traces.

Definition 1. *A trace graph G is a tuple (N, E, wit) where $N \subseteq \mathcal{B}$ is a set of nodes, $E \subseteq N \times N$ is a set of edges, and $wit : E \rightarrow \mathcal{T} \times \mathbb{N}$ is a function called the* witness *function mapping to each edge a pair consisting of a trace and a block index. In addition, the witness function wit is such that for each $e = (b, b') \in E$, $t \in \mathcal{T}$ and $j \in \mathbb{N}$ where $wit(e) = (t, j)$, it holds that $t[j-1] = b$ and $t[j] = b'$.*

Intuitively, a trace graph can be used to represent a set of execution traces obtained so far during a run, like a snapshot of the run. In this structure, each block is uniquely represented by a node. An edge is used as a *witness* (wit) of the trace and corresponding position where the block, represented by the edge target node, has occurred. More precisely, given a set of traces $T \subseteq \mathcal{T}$, a good representative trace graph must account for at least each block in T. This can be fulfilled by choosing the set of nodes to include the set of all blocks from the traces in T, and the set of edges to account for all pairs of blocks that occur consecutively in a trace from T. Observe that for the witness function there may be different traces that fulfill the condition of Definition 1, for example in case the traces overlap. Now the question is which trace to use in the function definition. To handle this, we assume that we are given a total order on the set T.

Definition 2. *For a set of traces $T \subseteq \mathcal{T}$ equipped with a total order $\preceq$, the* induced trace graph of T with respect to $\preceq$ *denoted by $G(T)$ is the trace graph (N, E, wit) where:*

- $N := \{b \in \mathcal{B} | \exists t \in T : b \in t\}$,
- $E := \{(b, b') \in \mathcal{B} \times \mathcal{B} | \exists t \in T : bb' \sqsubseteq t\}$, *and*
- $\forall e = (b, b') \in E$, $wit(e) := (t, j)$ *where* $t := \min\{t' \in T | bb' \sqsubseteq t'\}$ *(w.r.t. $\preceq$) and* $j := \min\{1 \leq k < |t| | t[k] = b \wedge t[k+1] = b'\}$.

Think of a total order on a set of traces as a way of arranging the traces in a specific sequence so that the witness function can be uniquely defined in the induced trace graph. If the traces are rearranged, for example by changing the order in which the inputs are fed to the program, the set of nodes and edges in the induced graphs are not affected. The trace order change only affects the witness function. In particular, rearranging overlapping traces changes the witness function value for the edges in scope of that overlap. This is since by definition

only the minimal trace, w.r.t. the global ordering, is used in the witness function definition a witness as captured in the following proposition.

This is relevant because it shows that the data structure captures the notion of *what* has happened, but is not bogged down by details about in which order they happened. Specifically, it allows strategies to be defined in terms of what has happened and what is possible, which seems more meaningful compared to in which order the information was collected. The following proposition formalizes the notion.

Proposition 1. *For a set of traces $T \subseteq \mathcal{T}$ and two total orders $\preceq_1$ and $\preceq_2$ on T, the induced graphs of T, G_1 and G_2 w.r.t. $\preceq_1$ and $\preceq_2$ are equal up-to an order automorphism.*

This property is useful for a distributed framework such ours, since regardless of which framework settings are used to analyze a program, the resulting trace graph is canonical in the sense of the previous proposition.

The order in our framework is that in which the traces are generated and we let $\preceq$ denote this order in the remainder of the section. Later on we will describe how graphs induced by this order can be used to define different selection strategies. What remains to do now is to devise an efficient method to construct such graphs incrementally so that it can be integrated in our framework.

3.3 Graph Construction

Assume a set of traces $T \subseteq \mathcal{T}$ and the induced trace graph $G(T) := (N, E, wit)$. Given a trace t, Algorithm 2 computes the graph induced by $T' := T \cup \{t\}$ where it is assumed that t is newly generated, i.e., t is a maximal element w.r.t. $\preceq$ in T'. The resulting trace graph is denoted by $G' := (N', E', wit')$.

Algorithm 2. Trace graph update algorithm

Input: A trace graph $G := (N, E, wit)$ induced by some set of traces $T \subseteq \mathcal{T}$ and a newly generated trace t

Output: The trace graph induced by $T \cup \{t\}$

1: $(n, N', E', wit') \leftarrow (\perp, N, E, wit)$
2: **for** $j = 1 \ldots |t|$ **do**
3:     **if** $t[j] \notin N'$ **then**
4:         add $t[j]$ to N'
5:     **end if**
6:     **if** $n \neq \perp \wedge (n, t[j]) \notin E'$ **then**
7:         add $e := (n, t[j])$ to E
8:         $wit'(e) \leftarrow (t, j)$
9:     **end if**
10:     $n \leftarrow t[j]$
11: **end for**
12: **return** $G' := (N', E', wit')$

Initially, G' is defined by copying the input graph G. The main loop of the algorithm iterates through the instruction blocks of the input trace t in their sequence order. The variable n, which is initially undefined, is used in the loop to keep track of the previous taint block in t. In each iteration j of the loop, first the block $t[j]$ is checked against the set N' and possibly added to it (l.3–5). Then, the edge $e := (n, t[j])$ is checked against the set E' and possibly added to it as well. In particular, when the edge e is added, then the function wit' is defined at e to be (t, j) (l.6–9). The algorithm has a linear time complexity in the length of the input trace $\mathcal{O}(|t|)$.

Proposition 2. *Given a set of traces $T \subseteq \mathcal{T}$, the induced $G(T)$, and a trace t such that t is the maximal element w.r.t. $\preceq$ of the set $T \cup \{t\}$, Algorithm 2 computes $G(T \cup t)$ w.r.t $\preceq$.*

3.4 Task Priorities

Another feature in the framework is the priority computation performed by the getPriority function in Algorithm 1. A common measure is usually based on the number of new instructions not encountered in previously processed traces. In our case, we adopt a similar approach based on the trace graph structure.

Given a set of traces $T \subseteq \mathcal{T}$ and a trace $t \in \mathcal{T}$, we define first two basic measures on t w.r.t. to T which we will later combine to define the main measure used in our framework. First, we define $nd(t, T)$ by:

$$nd(t, t') := |\{b \in t | \nexists t' \in T : b \in t'\}|.$$

Intuitively, nd counts the number of blocks occurring only in t. Second, we let $ed(t, T)$ denote the following:

$$ed(t, T) := |\{j | \exists t.t[j] \in T \wedge \forall t' \in T.t' \neq t : t'[j] \neq t[j]\}|.$$

This measure counts the number of new positions in which some of the blocks in the traces from T, occur in t. We are now ready to define the measure we used.

Definition 3. *Given a set of traces $T \subseteq \mathcal{T}$ and a trace t, the* rank *of t w.r.t. T denoted by $rk(t, T)$ is defined by*

$$rk(t, T) := nd(t, T) + ed(t, T).$$

Although, the computation of the rank can be fully integrated in the trace graph update algorithm, we choose to keep it separate in Algorithm 3 for clarity of the presentation. In this algorithm, the variables nd and ed are used to compute the measures $nd(t, T)$ and $ed(t, traces)$ respectively. The main loop iterates through the blocks of the input trace in their sequence order. The loop code block can be divided into two parts where in the first part (l.3–5) nd is updated and in the second one (l.6–15). Like in Algorithm 2, the variable n is used to keep track of the previous taint block in t.

Algorithm 3. Rank computation algorithm

Input: A trace graph $G := (N, E, wit)$ induced by some set of traces $T \subseteq \mathcal{T}$ and a newly generated trace t
Output: The rank of t w.r.t. T

```
 1: (n, nd, ed) ← (⊥, 0, 0)
 2: for j = 1 ... |t| do
 3:     if t[j] ∉ N then
 4:         nd ← nd + 1
 5:     end if
 6:     if n = ⊥ ∨ (n, t[j]) ∉ E then
 7:         v ← 1
 8:         for e ∈ E where wit(e) := (t′, k) do
 9:             if k = j ∧ t′[k] = t[j] then
10:                 v ← 0
11:                 break
12:             end if
13:         end for
14:         ed ← ed + v
15:     end if
16:     n ← t[j]
17: end for
18: return nd + ed
```

In each iteration j of the loop, first the taint block $t[j]$ is checked against the set n and nd is updated accordingly. Then, each edge $e \in E$, where $wit := (t', k)$ for some trace $t' \in t$ and $k \in \mathbb{N}$, is considered in order to check whether the current taint block $t[j]$ already occurs in the same position $(k = j)$ in some other trace t' from T. In case it does not, then the increment variable v is not reset and ed is updated accordingly.

Proposition 3. *Given a set of traces $T \subseteq \mathcal{T}$, the induced $G(T)$, and a trace $t \in \mathcal{T}$, Algorithm 3 computes $rk(t, T)$.*

4 Framework Features

We highlight some of the main features of OpenSAW including the choice of the underlying tools, the support for user-defined branch selection strategies, and the progress visualization web interface. An implicit feature is the choice of implementation language. OpenSAW is implemented in Python, a popular programming language. Hopefully, this will make it easier for users to write their own modules and extend the framework.

4.1 Choice of the Underlying Tools

OpenSAW uses a modular execution engine that is responsible for extracting execution traces and generating new inputs. The interface between this engine

and OpenSAW itself is generic enough so that it is easy to plug-in other engines. The engine currently supported in OpenSAW uses Intel's PIN [17] in combination with BAP [4]. For symbolic execution OpenSAW relies on BAP's iltrans tool. For constraint solving STP [9] is used.

4.2 User-Defined Strategies

Strategies are central in OpenSAW. They allow the user to steer the exploration of the executable. Strategies could be generic and based on common graph search algorithms. They could also be program specific and based for example on instruction addresses. OpenSAW's strategies enable users to control both the order in which tasks are handled and which branches of a trace to examine.

OpenSAW comes with some built-in strategies and offers support for user-defined ones. The built-in strategies include basic operations. For example, operations for handling redundancies by skipping analyzed branches, trying to improve coverage by only analyzing branches if they have exits that have never been taken, and trying to avoid loops by only analyzing each branch once per trace are included. They also cover graph search based ones such as Depth-first, Breadth-first and Generational search. In addition, OpenSAW has two built-in meta strategies for sequential and parallel composition of strategies. A sequential strategy chains the effect of its operand strategies while a parallel strategy rotates among them. The command line interface allows the user to choose and compose freely the built-in strategies.

In general, all the strategies in OpenSAW are derived from an abstract superclass with several callback methods. This superclass defines the strategy interface in OpenSAW. The callbacks offer hooks in the framework that are useful during the run. Each callback method is bound to a particular event so that it is only called when the corresponding event takes place. For example, there is a callback method for the generation of a new input, the extraction of a new trace, the failure of the constraint solver on a particular branch, etc. Users can extend the framework with other strategies as long as they implement the interface.

4.3 Progress Visualization

OpenSAW is shipped with a web interface for progress visualization and shown in the figures below. Figure 2 corresponds to the trace graph view. This view is animated so that the user can see the effect of the trace graph updates during the framework run. Figure 3 shows the statistics view. Among the component in this view is the pie graph illustrating how much of the overall runtime each of the underlying tool accounts for. Other components in this view include the number and type of crashes and a chart illustrating the number of visited branches over time.

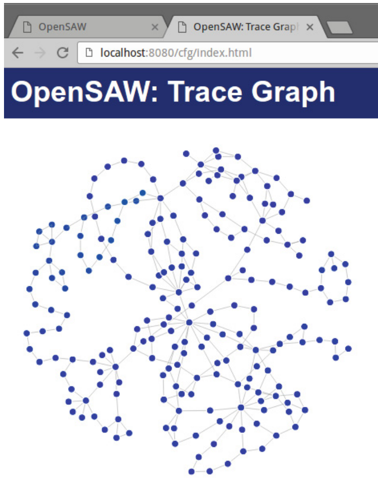

Fig. 2. OpenSAW features a trace graph view that visualizes the generation of the trace graph

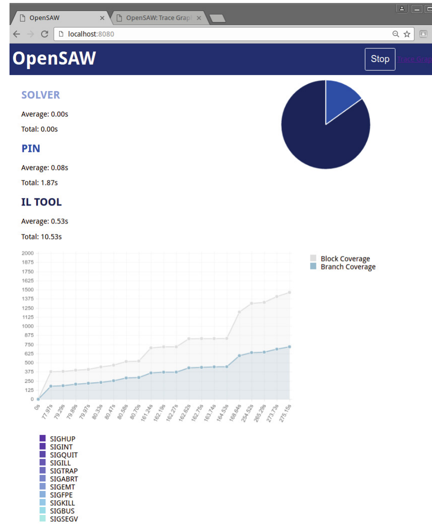

Fig. 3. OpenSAW features a statistics view that, for example, contains a graph of visited branches over time

5 Experiments

We have tested OpenSAW on the binaries used in DARPAS Cyber Grand Challenge (CGC) Qualification round [1]. CGC was a challenge developed to test the ability of cyber reasoning systems to find, prove and patch vulnerabilities in programs. We chose this test set because the programs are complex, contain vulnerabilities based on real-world-bugs and are written by people with different background and skill. Additionally we have evaluated OpenSAW on a codec used in production. By comparison to the CGC tests, the codec is much larger, runs on a real Linux system and requires inputs of much larger size.

5.1 DARPA CGC

The DARPA CGC qualification round consisted of 131 vulnerable binaries. These binaries run on a system called DECREE (DARPA Experimental Cyber Research Evaluation Environment) This is a system built on Linux but with only seven different syscalls. These allow I/O, memory allocation, randomness and program termination. This limitation leads to a small and well defined environment, which allows developers of analysis tools to focus on the analysis and not on the internals and quirks of the complete set of Linux syscalls.

We were able to execute OpenSAW on 126 of the 131 CGC binaries. Five binaries were omitted because they use inter-process communication and currently OpenSAW can only analyze one binary at a time. We let OpenSAW execute with the generational search strategy for 30 min per binary. If OpenSAW

had not exhaustively searched the input space of the binary within this time we aborted the search and continued with the next binary. For each binary the initial input consisted of the same 10 kB random data. The size of this initial input was chosen with the assumption that all binaries could be crashed with some input of this size.

The testing was done in a virtual machine running on a four core i7-4800 MQ CPU with 2.70 GHz and support for eight threads. The host machine only ran the virtual machine and assigned 15 GB of memory and four cores to it. Using this setup, OpenSAW found seven reproducible crashes that were not caused by the initial input.

5.2 Production Code

We also tested OpenSAW on a codec used in production. In contrast to the CGC binaries, the source code of the codec has 200 k lines. In addition it requires inputs of sizes between 0.5–120 kB to achieve high coverage of the code. The codec consists of two binaries: an encoder and a decoder. The initial input files for the encoder and the decoder were valid inputs of 46 kB and 120 kB in size respectively. It is worth noting that the codec had already been tested internally with AFL and presumably also by external users. As AFL was already in use internally we have compared OpenSAW with AFL. The tools are also similar as they both handle programs that use a single file as input and also in that neither requires any modification of the tested program or any additional wrapper code.

We ran two instances of OpenSAW simultaneously during 96 h, one on the encoder and the other one on the decoder on the same system setup as DARPA CGC. We also ran AFL on the codec for the same amount of time, on the same system setup, with multi-threaded AFL running for both the encoder and decoder. The result of these runs can be seen in Table 1.

The inputs generated by AFL did not identify any bugs. This is probably due to the fact that all the bugs revealed earlier by AFL have been already corrected in the version we were testing.

OpenSAW generated a bug finding input after 40 h. This bug was identified by executing the generated inputs on the codec with additional error detection in place. In comparison with AFL, OpenSAW also achieved better code coverage with fewer test cases.

Table 1. Results of running OpenSAW and AFL for 96 h each on the codec.

	Inputs	Bugs found	Function coverage in %	Line coverage in %
AFL	42 775	0	61.0	51.0
OpenSAW	464	1	66.5	53.3

6 Conclusion

Summary. We have presented OpenSAW, a new framework for white-box fuzzing. OpenSAWstrives to be open, flexible and agnostic to the underlying tools and techniques such as for symbolic execution, trace analysis and constraint solving. In fact, it could be used as a test platform for experimenting with such tools. We have addressed the issue of branch selection. For that purpose, we have defined the trace graph structure and generalized the concept of selection strategies to functions over trace graphs. This is one of the central features in OpenSAW which offers a large catalog of built-in strategies and support for user-defined ones. OpenSAW aims to be a flexible and efficient testing tool that can be scaled in or scaled out depending on the available computation power. In order to achieve this particular goal, we have addressed the issue of task prioritization and devised an efficient trace ranking algorithm fully integrated with the trace graph update procedure. We have tested OpenSAW successfully on an external benchmark and on an internal production code. In particular, the analysis of the production code did discover a new bug not revealed earlier by the testing process in place.

Future Work. There is potential for many research directions with OpenSAW. In terms of use cases, it could be interesting to use strategies that steer the search towards specific parts of the program binary. This would be a useful feature for example for restricting the testing during upgrades. In relation to strategies, another interesting direction could be to use machine learning to define good strategies. This could be based on observing the program behavior during normal operations and then developing a strategy that focuses the search along less common paths. Another example could be based on analyzing a large data set of OpenSAW runs over similar types of programs in order to identify crash patterns. In terms of extensions, it could be worth looking into how to integrate static analysis in the framework to further tune the search. For example, what would be the benefit when starting from an approximate CFG provided by a static analyzer. If used properly, such information could potentially reduce the size of the trace graph and make the search converge faster.

Reflections. Our industry is extremely heterogeneous in terms of software and hardware platforms. OpenSAW will always benefit from advances in symbolic execution techniques addressing for example the support of multi-threading and floating point computation. OpenSAW is good for testing in some niches and has proven to be a useful complement to the testing process for at least one. Therefore, this will only drive forward our quest in promoting and developing such technology.

References

1. DARPA Cyber Grand Challenge Competitor Portal. http://archive.darpa.mil/CyberGrandChallenge_CompetitorSite/
2. The international SAT Competitions web page. http://www.satcompetition.org/
3. Avgerinos, T., Rebert, A., Cha, S.K., Brumley, D.: Enhancing symbolic execution with veritesting. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 1083–1094. ACM, New York (2014)
4. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 463–469. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22110-1_37
5. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) OSDI, pp. 209–224. USENIX Association (2008)
6. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Juels, A., Wright, R.N., di Vimercati, S.D.C. (eds.) Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, October 30–November 3, 2006, pp. 322–335. ACM, Alexandria (2006)
7. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: IEEE Symposium on Security and Privacy, pp. 380–394. IEEE Computer Society (2012)
8. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A platform for in-vivo multi-path analysis of software systems. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI, pp. 265–278. ACM, New York (2011)
9. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007). doi:10.1007/978-3-540-73368-3_52
10. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 213–223. ACM, New York (2005)
11. Godefroid, P., Levin, M.Y., Molnar, D.: Sage: Whitebox fuzzing for security testing. Queue **10**(1), 20:20–20:27 (2012)
12. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: NDSS. The Internet Society (2008)
13. Haller, I., Slowinska, A., Neugschwandtner, M., Bos, H.: Dowser: A guided fuzzer for finding buffer overflow vulnerabilities. In: login: The USENIX Magazine. vol. 38(6), December 2013
14. Haller, I., Slowinska, A., Neugschwandtner, M., Bos, H.: Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In: Proceedings of the 22nd USENIX Conference on Security, SEC 2013, Berkeley, CA, USA, pp. 49–64. USENIX Association (2013)
15. Khurshid, S., PĂsĂreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003). doi:10.1007/3-540-36577-X_40
16. Lanzi, A., Martignoni, L., Monga, M., Paleari, R.: A smart fuzzer for x86 executables. In: Proceedings of the Third International Workshop on Software Engineering for Secure Systems, SESS 2007, p. 7. IEEE Computer Society, Washington, DC (2007)

17. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 190–200. ACM, New York (2005)
18. Ramos, D.A., Engler, D.: Under-constrained symbolic execution: Correctness checking for real code. In: Proceedings of the 24th USENIX Conference on Security Symposium, SEC 2015, pp. 49–64. USENIX Association, Berkeley (2015)
19. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, pp. 263–272. ACM, New York (2005)
20. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmalice- automatic detection of authentication bypass vulnerabilities in binary firmware (2015)
21. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Krugel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS (2016)
22. Sutton, M., Greene, A., Amini, P.: Fuzzing Brute Force Vulnerability Discovery. Pearson Education, Upper Saddle River (2007)
23. Zalewski, M.: American fuzzy lop. http://lcamtuf.coredump.cx/afl/