Symbolic assertion mining for security validation

Alessandro Danese
Department of Computer Science
University of Verona, Italy
Email: alessandro.danese@univr.it

Valeria Bertacco
Computer Science and Engineering
University of Michigan, Ann Arbor, MI, USA
Email: valeria@umich.edu

Graziano Pravadelli
Department of Computer Science
University of Verona, Italy
Email: graziano.pravadelli@univr.it

***Abstract*—This paper presents *DOVE*, a validation framework to identify points of vulnerability inside IP firmwares. The framework relies on the symbolic simulation of the firmware to search for corner cases in its computational paths that may hide vulnerabilities. Then, *DOVE* automatically mine a compact set of formal assertions representing these unlikely paths to guide the analysis of the verification engineers. Experimental results on two case studies show the effectiveness of the generated assertions in pinpointing actual vulnerabilities and its efficiency in terms of execution time.**

I. Introduction

In the past decade, the number of firmware attacks has been on the rise [1], [2]. For instance, erroneous hardware configurations let attackers set protected memory locations as writeable [3]; vulnerable update routines allowed the execution of malicious code [4]; and vulnerable interrupt handlers were exploited to attack a firmware by performing operations when the CPU was in the most privileged execution mode [5].

Consequently, more sophisticated validation techniques and tools are necessary to guarantee an effective identification of firmware vulnerabilities. Unfortunately, an exhaustive formal validation of the whole system is not any longer feasible for nowadays complex embedded systems. Then, verification engineers are more and more required to prioritize the validation effort to target the most exercised and vulnerable features of a design. However, this collides with the intractable diversity of firmware vulnerabilities, which makes their detection a very challenging issue. As a consequence, vulnerabilities hidden in unlikely execution paths risk escaping the validation.

As many classes of vulnerabilities are difficult to find without simulation, but exhaustiveness of the analysis is also important, recent works have combined symbolic simulation and assertion checking in order to verify firmware execution flows (see Section VI). These approaches rely upon a user-defined set of formal assertions describing behaviours the firmware should not implement. Such assertions, which are generally derived from a set of security requirements written in natural language, are first turned into checkers, and then verified in each execution path of the firmware. Unfortunately, the definition of assertions is a difficult and error-prone manual task. Omitting the definition of an assertion exposes to the risk of an incomplete validation process, possibly leading to the incapability of detecting actual vulnerabilities.

To overcome the manual definition of assertions and the related risks, this paper presents the *DOVE* (Detection Of firmware VulnErabilities) framework. *DOVE* automatically generates formal assertions that identify the unlikely execution flows of a firmware, where security vulnerabilities can escape generic validation efforts. To achieve its goal, it relies upon symbolic simulation and assertion mining and it exploits an abstract hardware model of the system under validation. The generated assertions describe how the firmware manipulates the memory locations and the registers of the hardware model in its different execution flows. Assertions are ranked according to the easiness of traversing the associated execution paths, so that verification engineers are guided to primarily inspecting the most unlikely (and then difficult to check) scenarios.

The rest of the paper is organized as follows. Section II summarizes the background. Section III presents *DOVE*. Section IV describes how the user can set up a validation session by using *DOVE*. Section V presents experimental results. Section VI deals with related works. Finally, Section VII concludes the paper with final remarks.

II. Background

A *symbolic simulation* is a way for exploring all execution paths of a program. It works by considering symbolic values in specific locations of the program. A symbolic value represents all the feasible values that can be assigned to a variable. At the beginning of the simulation, an initial symbolic state is created. A symbolic state represents a running process having register file, stack, heap and program counter. If during the simulation a symbolic state encounters a conditional statement, then new states are generated to follow each execution path depending on the condition. For instance, let us consider the function shown in Fig. 1a with a symbolic variable a. At the beginning, the symbolic state 1 is created as reported in Fig. 1b. Then, states 2 and 5 are generated from state 1 because of the condition at line 4. Next, states 3 and 4 are generated from state 2 because of the condition at line 6. When a new symbolic state is generated, an *edge* is defined between the state that reached the condition and the new one. Each edge is labelled with the constraints that have to be satisfied to follow the path originated by the new symbolic state. For instance, the constraint $a < 40$ labels the edge between states 1 and 2, meanwhile $a \geq 40$ is the constraint between states 1 and 5. The assignment of a value x to a variable w that occurs in the symbolic simulation is called a *snapshot* of the variable w. For example, during the symbolic simulation of the function in Fig. 1a, we can observe four different snapshots for b, namely: $\{(b = 0); (b = 1); (b = 2); (b = 3)\}$, and one snapshot for c, namely: $\{(c = 9)\}$. A sequence of snapshots that occur

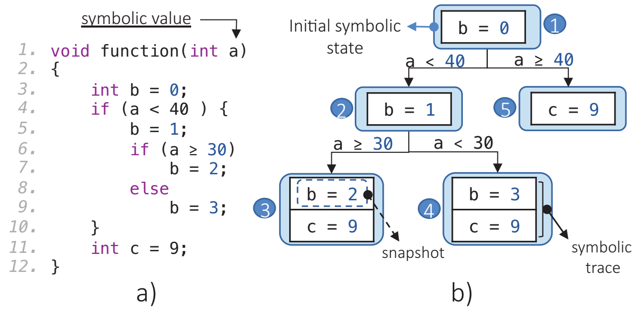

```
1.  void function(int a)
2.  {
3.      int b = 0;
4.      if (a < 40 ) {
5.          b = 1;
6.          if (a ≥ 30)
7.              b = 2;
8.          else
9.              b = 3;
10.     }
11.     int c = 9;
12. }
```

Fig. 1. A function and the corresponding symbolic tree.

during the execution of a symbolic state is called *symbolic trace*. The symbolic state 4 in Fig. 1b has the symbolic trace $\langle(b = 3); (c = 9)\rangle$. Throughout the paper we refer to the whole set of symbolic states and edges generated by the symbolic simulation as a unique data structure named *symbolic tree*.

III. The DOVE approach

Figure 2 presents the architectural view of the approach implemented in *DOVE*. The input parameters are a *firmware* in binary code and an *abstract model* of the hardware where the firmware is executed. The output is a set of *temporal assertions* highlighting corner cases that may hide security vulnerabilities of the firmware under verification. *DOVE* works in three phases:

(1) Symbolic simulation: The first phase consists of simulating the firmware with the purpose of maximizing its execution-path coverage. The result of this step is a *symbolic tree* tracing how the values of the registers and/or memory locations of the abstract hardware model change after the execution of each instruction of the firmware. The symbolic simulation of the firmware is performed by using KLEE [6]. The notions of symbolic simulation summarized in Section II are enough to understand the main contribution of our approach (represented by step 2 and step 3). Thus, details on this phase are omitted for lack of space.

(2) Probability mapper: In this phase, the *symbolic tree* is annotated with the probability of traversing each execution paths of the firmware. In particular, *DOVE* applies a weighted model counting-based approach to count the solutions that satisfy the constraints representing the conditions encountered along a path π. Afterwards, the number of solutions is used to compute the probability of executing π. Further details on this phase are provided in Section III-A.

(3) Assertion generator: The last phase is intended to generate formal assertions representing difficult-to-traverse execution paths that possibly hide security vulnerabilities and that escape traditional verification approaches. Assertions are ordered according to the probability of observing specific values in the registers and/or memory locations of the abstract hardware model during the execution of the firmware. Further details on this phase are provided in Section III-B.

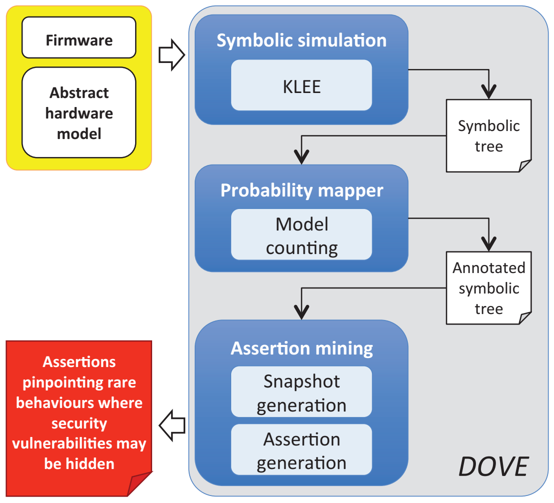


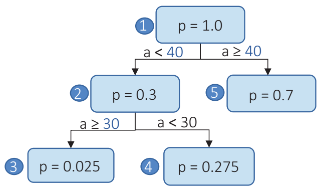
Fig. 2. Architectural view of DOVE.

Algorithm 1

```
1: function $probabilityMapper(v, C)$
2:     $p = computeProbability(v, C)$
3:     $setProbability(v, p)$ // assign probability $p$ to $v$
4:     for all $b$ in $edges(v)$ do
5:         $push(C, b.contraint)$ // add condition of $b$ to $C$
6:         $probabilityMapper(b.node, C)$ // recursion
7:         $pop(C)$
8:     end for
9: end function
```

State	Stack C	Probability
1	empty	1.0
2	a < 40	0.3
3	a < 40; a ≥ 30	0.025
4	a < 40; a < 30	0.275
5	a ≥ 40	0.7

a)



b)

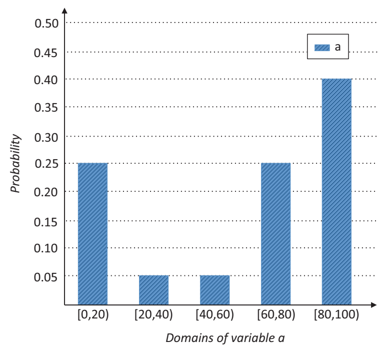
Fig. 3. Probability of reaching a symbolic state by satisfying the constraints along an execution path.

A. Probability Mapper

The probability mapper calculates the probability of reaching any of the symbolic states from the root of the symbolic tree. It traverses the symbolic tree with a depth-first based strategy. At each visited state v, a solver is applied to count the solutions of the conditions encountered along the execution path from the root to v. The counted solutions are then used to compute the probability of v. The result of this phase is an annotated symbolic tree reporting for each symbolic state its probability of being reached starting from the root.

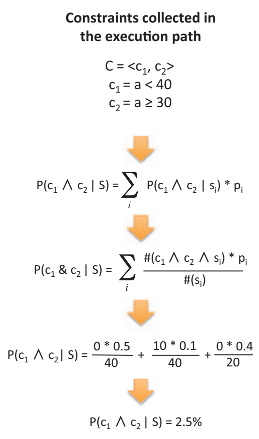
Algorithm 1 illustrates the pseudo-code of the function *probabilityMapper*. It takes as input parameters a symbolic state v, which initially is the root of the symbolic tree, and a stack of constraints C, which initially is empty. Then, it recursively computes the probability p of all the states in the tree by means of the function $computeProbability(C)$ (line 2), which is explained later in this section. As an example, let us consider to apply Algorithm 1 to the symbolic tree of Fig. 1b. Figure 3a shows, for each symbolic state, the constraints collected in the stack C, and the probability of reaching the state. Figure 3b reports the corresponding annotated symbolic tree.

The function *computeProbability* calculates the probability of satisfying a set of constraints $C = \langle c_1, \dots, c_n\rangle$ corresponding to an execution path π in the symbolic tree. Each c_i is an expression involving arithmetic-logic expressions among variables representing the condition to traverse the i-th edge of π. These variables are either primary inputs of the model under validation or they depend on primary inputs. It is worth noting that the values assumed by primary inputs could be not uniformly distributed. Thus, the probability of satisfying C is computed according to the actual probability distribution of the input variables, which is derived by performing a dynamic trace profiling of the system under validation. Based on this distribution, $DOVE$ computes the probability of satisfying C by exploiting the model-counting strategy proposed in [7]. Let D be the union of the domains of the input variables, and let $S = \langle(s_1), ..., (s_n)\rangle$ be a complete partition of D, where each s_i represents a different *input scenario*, i.e., the subset of the feasible values for the input variables characterized by the same probability p_i, with $\sum_i p_i = 1$. The value of p_i is derived by trace profiling. It represents the probability that the input values provided to the system at time t belong to s_i.



Scenario	Constraint	Probability
s_1	(0 ≤ a < 20) or (60 ≤ a < 80)	0.25 + 0.25 = 0.5
s_2	(20 ≤ a < 60)	0.05 + 0.05 = 0.1
s_3	(80 ≤ a < 100)	0.4

a)


Constraints collected in the execution path
C = <c1, c2>
c1 = a < 40
c2 = a ≥ 30
P(c1 ∧ c2 | S) = ∑i P(c1 ∧ c2 | si) * pi
P(c1 & c2 | S) = ∑i #(c1 ∧ c2 ∧ si) * pi / #(si)
P(c1 ∧ c2| S) = 0 * 0.5/40 + 10 * 0.1/40 + 0 * 0.4/20
P(c1 ∧ c2 | S) = 2.5%

b)

Fig. 4. Probability of satisfying a set of constraints given a set of scenarios partitioning the input space domain.

From the law of the total probability, the probability of satisfying the set of constraints C according to S can be computed as: $P(C|S) = \sum_i P(C|s_i) * p_i$. Then, by applying the law of conditional probability, we can rewrite the previous formula as $P(C|S) = \sum_i P(C \wedge s_i) * 1/P(s_i) * p_i$. Furthermore, $P(c)$ can be computed as $\#(c)/\#(D)$, where c is a constraint, and the operator $\#(.)$ returns the number of elements of D satisfying c. Thus, the previous formula can be finally rewritten as $P(C|S) = \sum_i \#(C \wedge s_i) * 1/\#(s_i) * p_i$.

Let us consider the set of constraints C listed on the top of Figure 4b, and the set of scenarios $S = \langle(s_1, p_1), (s_2, p_2), (s_3, p_3)\rangle$ derived from the probability distribution of the input values of the variable a (Figure 4a) as an example. By computing the conditional probability formula, the probability that C is satisfied in the first and third scenario is 0, since no value in either s_1 or s_3 satisfies all the constraints belonging to C. On the contrary, for the scenario s_2, ten values on forty satisfy all constraints in C. By weighting with the probability of s_2, namely p_2, we have $P(C|S) = 2.5\%$.

B. Assertion Generation

The goal of the assertion generator is to generate formal assertions pinpointing corner cases that may hide security vulnerabilities. The assertion generator works by using first the function $snapshotGenerator$ shown in Algorithm 2 to get a set of snapshots. Then, it uses the function $assertionMiner$ shown in Algorithm 3 to generate a temporal assertion for each snapshot. In this paper, we consider Linear Temporal Logic (LTL) assertions in the form *G(antecedent* $\rightarrow$ *consequent)*, where G is the LTL *always* operator[1], and *antecedent* and *consequent* may involve only *X*, namely the LTL *next* operator[2]. Moreover, *antecedent* is composed only of the constraints collected along an execution path of the symbolic tree. The *consequent* is a snapshot, namely a specific value in a register or a memory location of the hardware abstract model. As a shortcut, we will write $X[n](\alpha)$ to represent the application of n consecutive *next* operators to a formula α.

The function $snapshotGenerator$ gets as input parameters an annotated symbolic tree T, and the set W of traced variables during the symbolic simulation. At the beginning,

[1]Given a formula α, $G(\alpha)$ means that α is always true.

[2]Given a formula α, $X(\alpha)$ means that α is true at the next instant.

Algorithm 2

```
1:  function $snapshotGenerator(T, W)$
2:      $S = \emptyset$
3:      for all $w$ in $W$ do
4:          $S' = getSnapshots(T, w)$
5:          for all $s'$ in $S'$ do
6:              $p' = DFSAllPaths(T.root, s')$
7:              $s = annotateSnapshot(s', p')$
8:              $S = S \cup s$
9:          end for
10:     end for
11:     $sortByProbability(S)$
12:     return $S$
13: end function
```

Snapshots	Probability
(b=0)	1.0
(b=1)	0.3
(b=2)	0.025
(b=3)	0.275
(c=9)	0.7+ 0.275 + 0.025 = 1.0

a)

Ranking	Assertions
4	(b=0)
2	G (a < 40 → X[1](b=1))
1	G ((a < 40 & a ≥ 30) → X[2](b=2))
3	G ((a < 40 & a < 30) → X[2](b=3))
4	G (a ≥ 40 → X[1](c=9))

b)

Fig. 5. The probability of observing a snapshot and the corresponding generated assertion.

Algorithm 2 initializes the list of snapshots S with $\emptyset$ (line 2). Next, for each variable w of W (lines 3-10), it collects in S' the set of snapshots of w by visiting the tree T (line 4, $getSnapshots(T, w)$). Then, for each snapshot s' of S', Algorithm 2 gets the probability p' of observing s' during the simulation of the firmware by using the function *DFSAllPaths* (line 6). The intuitive idea is to reach all symbolic states belonging to different execution paths where the snapshot s' was traced for the first time. The probability p' of observing s' is, therefore, the sum of the probabilities of reaching the identified symbolic states. The function *DFSAllPaths* implements this strategy through a depth-first search algorithm. It gets as input parameters a symbolic state v, which is initially the root of the annotated symbolic tree T, and a snapshot s'. If s' is traced in v, then *DFSAllPaths* returns the probability p of v. On the contrary, *DFSAllPaths* forwards this search to each node reachable from v through a recursive call. If a leaf node has not the snapshot s', then *DFSAllPaths* returns the value 0. All returned values are then summed up and returned as final result. At the line 7, Algorithm 2 annotates the snapshot s' with the probability p' by using the function $annotateSnapshot(s', p')$. The annotated snapshot s is then inserted into the list S (line 8). At the end of Algorithm 2, the list S of snapshots is sorted in accordance with their probabilities (line 11), and returned as result (line 12).

As an example, let us consider the annotated symbolic tree of Fig. 3b as input of Algorithm 2. Figure 5a shows, for each snapshot s, the probability of observing s during the simulation of the firmware.

The second step of the assertion generation phase is to generate an assertion for each snapshot. The function $assertionMiner$ works as shown in Algorithm 3. It gets as input parameters the annotated symbolic tree T and a list of snapshots S sorted by using the probability. At the beginning, Algorithm 3 initializes a list of assertions A with $\emptyset$ (line 2). Next, for each snapshot s in S (lines 3-8), it searches by using the function *BFSearch(T.root, s)* (line 4) a symbolic state v of T where the snapshot s was traced. To keep compact the antecedent of the assertions the function *BFSearch* implements a breadth-first search based strategy to identify the state s. In detail, the function defines at the beginning a frontier F of states, which initially contains only the root node of T. If

Algorithm 3

```
1: function $assertionMiner(T, S)$
2:     $A = \emptyset$
3:     for all $s$ in $S$ do
4:         $v = BFSearch(T.root, s)$
5:         $antecedent = getRevOrderConstraints(v, T.root)$
6:         $a = makeAssertion(antecedent, s)$
7:         $A = A \cup a$
8:     end for
9:     return $A$
10: end function
```

a state v of F traced the snapshot s, then v is returned as results. On the contrary, *BFSearch* enlarges F by collecting all the reachable states of T from a state already in F. The set F is continually enlarged as long as a state v that traced s in its symbolic trace is identified. The antecedent for the snapshot s is then generated by collecting in inverse order the constraints of the edges of the annotated symbolic tree T from the state v to the root (line 5). A new assertion having the snapshot as consequent is generated (line 6), and inserted into the list A (line 7). At the end, Algorithm 3 returns the list of generated assertions (line 9). As an example, let us consider the annotated symbolic tree of Fig. 3b, and the snapshots of Fig. 5a as inputs of Algorithm 3. Figure 5b shows, for each snapshot s, the generated assertion with the corresponding ranking value (the lowest is the rarest corner case). Let us consider the assertion ranked as the most rare corner case, e.g. $G((a < 40 \wedge a \geq 30) \rightarrow X[2](b = 2))$. This assertion reports: if the constraint $(a < 40) \wedge (a \geq 30)$ is satisfied, then variable b gets value 2 after 2 simulated instructions. This assertion belongs to an unlikely simulation flow since only with probability 2.5% the values of a can satisfy the antecedent of the assertion. The other assertions in Fig. 5b have instead a much higher probability because many distinct values of a satisfy their antecedents. The snapshot at the first row does not involve the variable a. It reports that immediately b has value 0 without any constraint. A verification engineer can then be addressed to investigate if the detected unlikely execution flow may hide a security vulnerability.

IV. Framework Setup

Figure 6 shows the simulation environment of the proposed framework. It consists of an ARM instruction set simulator (ISS), a firmware loader that copies the firmware binary code under validation into an internal memory of the hardware model, and a memory-mapped register interface (MMRI) that allows users specifying which memory address a register is mapped to (if any). Given a firmware implementation, the verification engineer is supposed to customize the following components of the framework before starting the firmware validation.

Memory mapped-register interface: if, for the scenario of interest, some registers are memory mapped, then the user has to implement the methods *writeMMRI* and *readMMRI* to allow the ISS accessing (in read and write mode) those registers during the symbolic simulation. Algorithm 4 shows an example of a *C++* implementation of the function *writeMMRI* for writing in a memory-mapped register.

Similarly, Algorithm 5 shows an example of a *C++* implementation of the function *readMMRI* for reading a memory-mapped register.

Symbolic values: To perform symbolic simulation of the firmware some memory locations and/or registers of the abstract hardware model have to be filled out with symbolic values. This part is the most crucial of the validation process

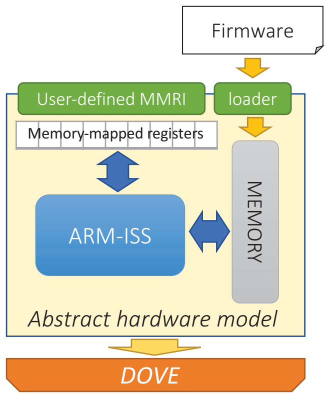


Fig. 6. Overview of the abstract hardware model.

as a too large number of symbolic variables can lead to the path explosion problem: the number of globally feasible paths is roughly exponential in the size of the whole system. On the contrary, an insufficient number of symbolic variables can lead to not showing the security vulnerability of the firmware since the unlikely execution flow is not simulated. With *DOVE* a user can perform a selective symbolic simulation targeting the firmware functionalities that are more exposed to attacks. In particular, it can address the framework to continually generate new symbolic values for only the memory locations that will be read by the functionalities under validation. *DOVE* provides the user with the *C++* function:

***void** fillOutSymbolicVariables(**Address** addresses).*

It gets as input parameter an array of memory addresses. *DOVE* will store a new symbolic value in each addressed register or memory location before executing an instruction of the firmware.

Traced values: To generate assertions some memory locations and/or registers of the abstract hardware model have to be traced. The definition of these locations is up to the user, which can drive *DOVE* to consider specific as well as generally-used registers and memory locations. *DOVE* provides the user with the *C++* functions:

Algorithm 4

```
1: bool $writeMMRI$(Address $a$, Data $d$, MMIO $*reg$) {
2:         switch (a) {
3:                 case 0x4000: {
4:                         reg[0] = d;
5:                         return true;
6:                 } case 0x4554: {
7:                         reg[1] = d;
8:                         return true;
9:                 } default:
10:                         return false;
11:         }
12: }
```

Algorithm 5

```
1: bool $readMMRI$(Address $a$, Data &$d$, MMIO $*reg$) {
2:         switch (a) {
3:                 case 0x4000: {
4:                         d = reg[0];
5:                         return true;
6:                 } case 0x4554: {
7:                         d = reg[1];
8:                         return true;
9:                 } default:
10:                         return false;
11:         }
12: }
```

void *trace(****Address*** *addresses);*
void *traceCPU(****Register*** *regIndices).*

The first function gets as input parameter an array of memory addresses, meanwhile the second gets an array of indices for the CPU registers.

V. Experimental evaluation

We evaluated the effectiveness and the efficiency of *DOVE* in two case studies concerning the validation of a memory protection mechanism and of an interrupt service handler. In both cases, the experimental results have been carried out on a 2.6 GHz Intel Core i5 processor equipped with 8 GByte of RAM and running Linux OS.

A. Case study 1: memory protection mechanism

The firmware we analysed acts as an interface between a memory-mapped IP and an upper-level software. The firmware reads values from the IP interface, then it elaborates a memory address on which it stores the read values. In this scenario, the memory location storing the firmware code is not writeable unless the flag $bioswe$ is set. Moreover, each attempt to change this flag causes an interrupt which resets the $bioswe$ at its default value, *i.e.* zero. In a correct execution flow, each value coming from the IP is then properly stored in a memory location, and, more importantly, any attempt to manipulate the firmware code itself has no effect. However, a security vulnerability located in the interrupt controller can expose the system to attacks. In fact, if interrupts can be disabled, then $bioswe$ is exposed to be set, and the firmware code in memory can be successfully overwritten afterwards, as reported in [3]. Consequently, if the firmware does not check the values provided to its input interface, an attacker can exploit it to first disable the memory protection mechanism, and then to write in a protected memory location.

We run *DOVE* in the above vulnerable context. We configured DOVE such that it considered symbolic values for the registers were the IP core interface was mapped. 30 assertions were generated. In Fig. 7 the generated assertions are represented by the blue points. They are ordered according to the probability of traversing their corresponding execution paths of the firmware. The assertion with the lowest probability (i.e., $1.4e{-}39$) is:
$a1 = G((X(offset = 25) \wedge X[3](data = 0) \wedge X[9](offset = 49) \wedge X[11](data = 1) \wedge X[17](23 > 300 + offset)) \rightarrow X[22](bioswe = 1))$.

By analysing the meaning of $a1$, we observe that it captures the situation where the values generated by the IP interface asks the firmware to: 1) disable the interrupts (when $offset$ is 25, the generated physical address hits gbl_smi_en); 2) enable the protected memory (similarly, $bioswe$ is hit when offset has value 49); 3) perform a write in its own code. All memory addresses satisfying the constraint $(23 > 300 + offset)$ hit the text memory. The behaviour captured by $a1$ actually highlights that the firmware can be exploited to attack the system as reported above. Other top ranked assertions ($a2$, $a3$) highlighted the same behaviour. In fact, they correspond to execution paths where the memory mapped registers are manipulated by the firmware in a different order and with different values. This proves *DOVE* was able to focus the attention on rare and dangerous execution paths representing the presence of a security vulnerability.

Subsequently, we removed the vulnerability by modifying the firmware such that it cannot write into the memory mapped register gbl_smi_en. Then we run *DOVE* again. This time

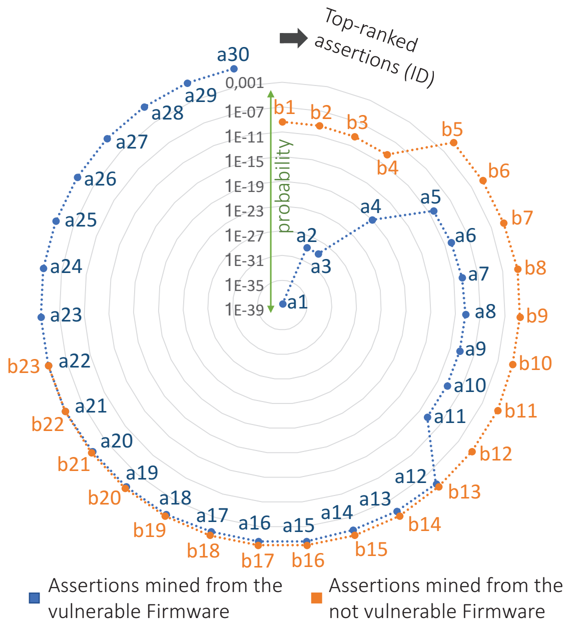


Fig. 7. Case study 1: Probabilities of assertions generated by *DOVE*, with (blue) and without (orange) firmware vulnerability.

DOVE generated 23 assertions represented by the orange points in Fig. 7. The probabilities of this new set of assertions is much higher compared with the probabilities of blue assertions. Thus, no extremely rare behaviour is highlighted this time. In addition, the top-ranked orange assertions (*b1*, ..., *b4*) still represent behaviours where the firmware tries to access memory-mapped registers, but none of them corresponds to a path where gbl_smi_en is modified.

B. Case study 2: interrupt service handler

Interrupt handlers are stored in a protected and inaccessible memory location. The base address of this location is stored in a CPU internal register (e.g., *SMBASE*). When an interrupt is triggered by the firmware, the CPU switches to the supervisor mode, copies the values of *SMBASE* in a location of the protected area, and starts fetching the instructions of the proper interrupt handler. When the handler returns, the CPU restores *SMBASE* by coping back its value from the protected area and switches to the user mode. In this scenario, a security attack can exploit a vulnerable implementation of an interrupt handler to execute unauthorized operations when the CPU is in a privileged mode as reported in [5]. In a not-attacking execution flow, the interrupt handler is supposed to write values in a buffer outside the protected area. However, if the interrupt handler does not check the correctness of values provided as inputs, an attacker can exploit this vulnerability to overwrite the value of *SMBASE* stored in the protected area. When a second interrupt then occurs, a memory location referred to a malicious code is read by the CPU owing to a different value in *SMBASE*.

We run *DOVE* in the above scenario and it generated 60 assertions involving the program counter (PC) of the CPU. Figure 8 shows in the x axes the memory addresses corresponding to the instructions pointed by the PC during the simulation, and in the y axes the probability that the PC points to those memory addresses. The top ranked assertions generated by *DOVE* pinpointed the corner cases in the execution flow of the firmware where unauthorized operations are performed, thus guiding the verification engineers towards

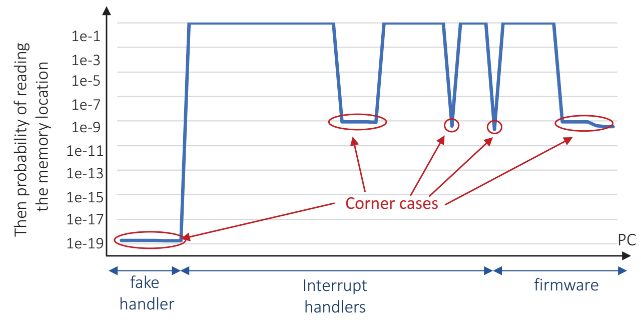


Fig. 8. Probability of reading the memory location addressed by the PC.

TABLE I
CASE STUDY 1: SYMBOLIC STATES, ASSERTIONS AND EXECUTION TIME.

Instr.	States	Assertions	Sim. time	P. mapper time	Gen. time	Total time
5	1	9	< 0.1 s.	< 0.1 s.	< 0.1 s.	< 0.1 s.
10	7	17	< 0.1 s.	0.2 s.	< 0.1 s.	0.2 s.
15	37	21	0.6 s.	1.3 s.	< 0.1 s.	1.9 s.
20	37	23	0.8 s.	1.3 s.	< 0.1 s.	2.1 s.
25	188	27	3.2 s.	8.7 s.	< 0.1 s.	11.1 s.
30	188	30	4.3 s.	8.9 s.	< 0.1 s.	13.2 s.

suspicious behaviour that may correspond to security holes. In particular, the assertion associated to the most unlikely execution path (probability= $1.4e{-}18$) was:
$G(X[37](mem[0] = 0) \wedge X[48](mem[4] = 0x4108) \rightarrow X[68](PC = 0x3910))$.

This assertion reports how specific values processed by the interrupt handler allow the CPU to fetch an instruction from the memory address $0x3910$, where we stored a fake (malicious) handler to reset the value of $SMBASE$.

C. DOVE scalability

Table I and Table II report information to analyse the scalability of the approach implemented in *DOVE*. In particular, their columns refer, from left to right, to the number of instructions executed by the firmware and consequently to the number of states generated by the symbolic simulation, the number of assertions generated by *DOVE*, the time required by the three phases of *DOVE*, and the total execution time concerning for the two use cases.

As expected, the higher is the number of instructions executed by the firmware, the larger is the set of generated symbolic states. The number of generated assertions increased as well, nonetheless the final set of assertions is small and generated in a few seconds. For the first case study we stopped when no more symbolic state was generated, while in the second case study we terminated the simulation when no new assertion was mined.

VI. RELATED WORK

FIE [8] is a platform built on top of KLEE for detecting bugs in small firmware programs for the MSP4030 family of microcontrollers. It currently supports finding only two types of bug: memory safety violations and peripheral-misuse errors. S^2E [9] is a platform built on top of the QEMU virtual machine [10] and the KLEE symbolic execution engine. It scales to large real systems by selectively executing symbolically only those parts of a system that are of interest to the tests. In [11] a tool is proposed to specifically detect unauthorized read accesses performed by interrupt handlers. *DDT* is instead a system for testing closed-source binary device drivers against undesired behaviours, like race conditions, memory errors, resource leaks [12]. It provides a default set of checkers that can be extended to specify both safety and liveness properties, but it cannot generate assertions automatically. Finally,

TABLE II
CASE STUDY 2: SYMBOLIC STATES, ASSERTIONS AND EXECUTION TIME.

Instr.	States	Assertions	Sim. time	P. mapper time	Gen. time	Total time
25	1	19	< 0.1 s.	< 0.1 s.	< 0.1 s.	< 0.1 s.
50	11	41	0.1 s.	0.2 s.	< 0.1 s.	0.3 s.
75	95	57	1.1 s.	2.4 s.	< 0.1 s.	3.5 s.
100	211	58	2.3 s.	6.8 s.	< 0.1 s.	9.1 s.
125	340	60	7.0 s.	12.5 s.	< 0.1 s.	19.5 s.
150	456	60	10.8 s.	20.5 s.	< 0.1 s.	31.4 s.
175	594	60	19.6 s.	31.3 s.	< 0.1 s.	51.0 s.
200	3343	60	63.7 s.	170.3 s.	< 0.1 s.	234.3 s.

KleeNet [13] and *T-Check* [14] are tools that use symbolic analysis to generate test cases for sensor networks and to find safety and liveness errors in sensor network applications running on TinyOS. With respect to the above approaches, *DOVE* is not focused on a specific kind of vulnerability, and it automatically generates assertions highlighting corner cases in firmware execution to allow verification engineers precisely focussing on suspicious behaviours.

VII. CONCLUSION

We proposed a framework to detect firmware vulnerabilities that can be exploited by an attacker to break the system security. *DOVE* is based on a symbolic simulation engine that exhaustively explores the computational paths of the firmware and provides the verification engineers with a ranked set of assertions. These assertions describe corner cases in the firmware execution where vulnerabilities could remain undetected by applying traditional verification approaches. *DOVE* effectiveness and efficiency have been evaluated in two actual case studies. In both cases, *DOVE* was able to highlight the vulnerabilities in a few seconds by generating a compact set of assertions. The set up of the verification process performed with *DOVE* required to add a few lines of code inside an ARM-based abstract model of the target hardware.

Acknowledgements. This work was supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA.

REFERENCES

[1] 2017 threats predictions. McAfee Labs. [Online]. Available: https://www.mcafee.com/us/resources/reports/rp-threats-predictions-2017.pdf
[2] C. Kallenberg and X. Kovah, *How Many Million BIOSes Would you Like to Infect?* CanSecWest, 2015.
[3] C. Kallenberg, S. Cornwell, X. Kovah, and J. Butterworth, "Setup for failure: defeating secure boot," in *Proc. of SyScan*, 2014.
[4] C. Kallenberg, J. Butterworth, X. Kovah, and C. Cornwell, "Defeating signed bios enforcement," MITRE, Tech. Rep., 2013.
[5] L. Duflot, O. Levillain, B. Morin, and O. Grumelard, "System management mode design and security issues," *IT Defense*, February 2010.
[6] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of USENIX OSDI*, vol. 8, 2008, pp. 209–224.
[7] A. Filieri, C. S. Păsăreanu, and W. Visser, "Reliability analysis in symbolic pathfinder," in *Proc. of IEEE ICSE*, 2013, pp. 622–631.
[8] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Proc. of USENIX Security*, 2013, pp. 463–478.
[9] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 265–278, 2011.
[10] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. of USENIX Annual Technical Conference*, 2005, pp. 41–46.
[11] O. Bazhaniuk, J. Loucaides, L. Rosenbaum, M. R. Tuttle, and V. Zimmer, "Symbolic execution for bios security," in *Proc. of USENIX WOOT*, 2015.
[12] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing closed-source binary device drivers with ddt," in *Proc. of USENIX Annual Technical Conference*, 2010.
[13] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment," in *Proc. of ACM/IEEE IPSN*, 2010, pp. 186–196.
[14] P. Li and J. Regehr, "T-check: bug finding for sensor networks," in *Proc. of ACM/IEEE IPSN*, 2010, pp. 174–185.