# Efficient Program Exploration by Input Fuzzing

Guillaume Bonfante[1,2], Jean-Yves Marion[1,2], and Thanh Dinh Ta[1,2]

[1] Université de Lorraine
[2] CNRS - INRIA Nancy

**Abstract.** One of the issues of a malware detection service is to update its database. For that, an analysis of new samples must be performed. Usually, one tries to replay the behavior of malware in a safe environment. But, a bot sample may activate a malicious function only if it receives some particular input from its command and control server. The game is to find inputs which activate all relevant branches in a bot binary in order to retrieve its malicious behaviors. From a larger viewpoint, this problem is an aggregation of the program exploration and the message format extraction problem, both of them captures many active researches. This is a work in progress in which we try a new approach to code coverage relying on input tainting.

## 1 Introduction

We extract from the honeypot of our High Security Lab[3] many suspicious samples. A large part of them are bots or trojans. Indeed, we observe, by monitoring the network traffic, their communication with external hosts. Our aim is to find the payloads of these programs. To do that, we try to find some run which will reveal the malware functionalities. We work both by dynamic analysis–to avoid some obfuscation techniques–and by static analysis–for a full coverage of the binary code. In an attempt to extract automatically properties of malicious samples, we relate our work to existing researches on several domains: *execution trace coverage* [6, 9], *symbolic execution* [1, 10] and *protocol message format extraction* [5, 11, 12].

**Current researches** The *execution trace coverage* aims to cover the program's behaviors as much as possible by observing its execution under different inputs. For example, the method discussed in [6] analyzes execution traces, manipulate them, and solve branch decision by input modification.

Usually, the *symbolic execution* is applied to programs with available sources. It aims to partition the program's input set into equivalence classes so that the inputs in the same class correspond to the same (up to some equivalence) execution trace. Normally, each class is characterized by a first-order logic formula whose variables are also some program's variables. The main difficulty in our case is that we have only access to a binary, and even worse an obfuscated one.

[3] http://lhs.loria.fr

The *protocol message format extraction* aims to restore the format of messages used by the program. Since there is no certainty about which format should the program follow, the current methods suppose that, for example, messages consist of fields with a static hierarchical structure, and dynamically analyze how the program consume messages to restore this structure, see [5] for instance.

**Our approach** Similar to the *symbolic execution*, we aim at partition the set of inputs into equivalence classes. However we have only the binary code of malware samples, so methods of the *execution trace coverage* will be applied. As a result, the equivalence classes of inputs are not characterized by logic formulas but some equivalence relations on the program's execution traces.

Moreover, the equivalence class partition can be used as a semantics of the program. For programs receiving messages as inputs, the partition on the set of inputs (i.e. received messages) contains also the information about the format of messages. Consequently, it can help constructing a foundation for the *message format extraction* problem without using any "artificial" supposition: the message's semantics is not independent from program, it should instead emerge naturally from the way the program consumes messages.

Finally, the partition-based semantics is close to the input/output semantics, which is always preserved in the code obfuscation techniques [8]. So the malware detection based on this semantics promises that it will be resilient to them.

**Contribution** The main contributions of the paper are as follows:

– By restating the classical program exploration problem under a more general viewpoint, we present a new program exploration method whose the technique is orthogonal to ones used in current methods.
– To experiment the technique, we present an implementation based on the Pin DBI framework [3]. The implementation does not require a whole system emulator for dynamic code instrumenting and state saving/restoring.
– We propose a semantics framework for the message format extraction problem. The framework extracts the format from an input partition obtained by inferring execution traces of a binary. As a result, one might expect to recover the semantics of communications between a command and control server and bots.

**Content** In section 2, we review the some related terminologies and state the problem. The algorithms are presented in section 3, the implementation and its experimental results are presented in section 4. In section 5, we discuss about the future improvement.

## 2 Program exploration

We first review some technical terms (limited to the `x86` architecture) to clear out important facts. We then describe and state the problem of interest.

**Terminology and background**

At loading time, a program gets its *static form*, that is a sequence of consecutive instructions. An *instruction ins* is a pair $\langle addr, dat \rangle$ where *addr* is the memory address and *dat* is the content stored at this address. The latter consists of an *operator* and some *operands*.

At running time, the instruction pointer (that indicates the address of the next executed instruction) can be diverted from the sequential order (as specified in the program's static form) by instructions like *function call*, *conditional* and *unconditional jump*. In this paper, we focus our attention to conditional jumps. A sequence (i.e. the executed-time ordered multi-set) of executed instructions is called an *execution trace*, or shortly a *trace*.

The program needs to react to the external environment, e.g. receiving inputs. In this case, its memory space is modified not only by its instructions but also by ones of external programs (which include the operating system). The change of execution from one to the other is realized by an *exception*.

*Example 1.* Following is a portion extracted from `libc`'s static form at loading time, the instruction's contents are displayed in the mnemonic form.

```
0x7f26bfa4c32d   mov eax, 0x5
0x7f26bfa4c332   syscall
0x7f26bfa4c334   cmp rax, 0xfffffffffffff000
0x7f26bfa4c33a   jnbe 0x7f26bfa4c354
0x7f26bfa4c33c   ret
```

Locates at `0x7f26bfa4c32d` is an *instruction* with the content `"mov eax, 0x5"`. Its operator is `mov`, the *explicit operands* are the register `eax` and the immediate value `0x5`, the *implicit operand* (which is not displayed) is `rflags`. The instruction $\langle$`0x7f26bfa4c332`, `syscall`$\rangle$ is an *exception*. One of its implicit operand is the register `rax`, it's value `0x5` actually corresponds to a `sys_open` system call. The *conditional jump* `jnbe 0x7f26bfa4c354` locates at `0x7f26bfa4c33a`. If the flag `CF` and `ZF` are both zero then it makes the next executed instruction be located at `0x7f26bfa4c354` instead of `0x7f26bfa4c33c` as shown in the static form. The values of `CF` and `ZF` are set by the execution of the instruction at `0x7f26bfa4c334` with content `cmp rax, 0xfffffffffffff000`.

A directed graph can be constructed from the execution trace as follows: the nodes are distinguished instructions, a directed edge $ins_1 \to ins_2$ is drawn for each pair $(ins_1, ins_2)$ of consecutive instructions in the trace. This graph is called *dynamic control flow graph* (abbr. DCFG).

Similarly, a directed graph can be constructed from the static form but directed edges $ins_1 \to ins_2$ are added for all possible target $ins_2$ of $ins_1$ (which is a function call or jump) where $ins_2$ is the instruction located at the target of $ins_1$. This graph is called *static control flow graph* (abbr. SCFG).

*Note 1.* Before going to state the problem, we note that the construction of DCFGs is always decidable but the construction of SCFG is undecidable in general. Moreover, the DCFG depends strictly on the execution trace (i.e. we can receive different DCFGs given different inputs) and is a multi-graph, while the SCFG is unique and is always a simple-graph. However, if we identify all edges of the same head and tail in a DCFG, then the DCFG is always a subgraph of the SCFG. Consequently, *by collecting and analyzing enough DCFGs, one might expect to approximate the SCFG.*

## Code coverage problem

The *dynamic analysis* relies on the study on dynamic traces or on the dynamic control flow graph. It is rapid and precise but terribly incomplete: examining on concrete execution instances of the program, it gives precise information for them, but hardly predicts what will happen in others. In contrast, the *static analysis* studies properties of programs without running them, i.e. it verifies the properties that are still true for all execution instances, then say, it is more complete than the dynamic analysis.

There are several good reasons to consider the dynamic approach. Some of them are stated in [13], but the main reason comes from the fact that malware are obfuscated programs so that their static control flow graph cannot be (in general) built precisely [7].

But the dynamic approach has disadvantage because of its incompleteness: some functions of the sample will not be executed (and then become invisible under the analysis) if particular external conditions (e.g. environment states, inputs, etc) are not satisfied; without knowledge about these functions we do not know for certain whether the sample is malicious or not. Consequently, a problem posed to improve the completeness is informally stated as follows:

*Problem 1 (Code coverage).* Given a program $P$ and some function $f$, how to construct algorithmically (i.e. automatically) an input $m$ so that when executing with the input, the program activates $f$.

Without any constraint on examined programs, the code coverage problem is undecidable. We are interested then in a more modest one.

*Problem 2 (**Branch coverage**).* Let $P$ be a program and $M$ be the finite set of inputs, given a dynamic trace:

$$Tr\,(P, m) = T_1.b.c.T_2$$

where $T_1, T_2$ are traces, $T_1$ is finite, $b$ and $c$ are instructions where $b$ is conditional jump, resulted from executing $P$ with an input $m \in M$. Find $m' \in M$ so that the trace resulted from executing $P$ with $m'$ has form:

$$Tr\,(P, m') = T_1.b.c'.T_2'$$

where $c' \neq c$ and $T_2'$ is some trace.

*Note 2.* In some sense, the branch coverage problem is more general than the code coverage because of a single conditional jump can be resolved multiple times on the execution trace. The following example illustrates this difference.

*Example 2.* Consider the traces $T$ and $T'$:

$$T = T_1.b.c.T_2.b.c'.T_3 \quad \text{and} \quad T' = T_1.b.c'.T_2.b.c.T_3$$

With respect to the SCFG, the conditional jump $b$ is fully covered: decisions to $c$ and to $c'$ are taken both, and the graph of $T$ and of $T'$ are identical. But with respect to the execution trace, $b$ is covered at the first occurrence (since two traces have the same prefix $T_1$) but not at the second one (since $T_1.b.c.T_2 \neq T_1.b.c'.T_2$).

## 3 Branch resolving algorithm

The algorithm proposed in this section aims to resolve the branch coverage problem in a practical context. In the following, we first restate the problem and then present the core ideas of our algorithm, the pseudo-codes (which are technical and maybe tedious to follow) are presented in appendix A.

*Problem 3 (Practical branch coverage).* Let $P$ be a program, $M$ be the finite set of inputs. Given an input $m \in M$, the execution of $P$ on $m$ results in a trace:

$$Tr\,(P, m) = T_1.b_1.c_1.T_2.b_2.c_2 \ldots$$

where $b_i, c_i$ are instructions, $b_i$ is a conditional jump, $T_i$ is a trace for $i = 1, 2 \ldots$. The problem is to find inputs $m_1, m_2, \ldots \in M$ so that when executing $P$ on them result in traces of the following corresponding forms:

$$\begin{aligned} Tr\,(P, m_1) &= T_1.b_1.c_1'.T_1' \\ Tr\,(P, m_2) &= T_1.b_1.c_1.T_2.b_2.c_2'.T_2' \\ &\ldots \end{aligned}$$

In other words, each $m_i$ should be the solution of the original branch coverage problem with respect to the trace $Tr\,(P, m)$ and the conditional jump $b_i$.

### Algorithm

The main algorithm is *branch resolving* together with auxiliary functions: *dynamic tainting* and *nearest checkpoint computation*. To present the core ideas, the following technical terms need to be clear.

*Checkpoint* A checkpoint stores the state of a program (e.g. values of processor's registers and of the program's memory space). We can replay the execution of the program by restoring its state to the one saved in the checkpoint.

*Coherent rollback* A rollback means restoring the program's state to a certain checkpoint. A coherent rollback means restoring the state to a certain checkpoint along with some modification on that state, so that the modification does not conflict with the state resulted from the execution before the checkpoint.

*Hyper-graph* A directed hyper-graph is a directed graph but an edge (named hyper-edge) can have multiple heads and tails.

**Branch resolving** Roughly speaking, we first execute the program on some arbitrary input and collect only conditional jumps (abbr. *branch*) that depends on the input. Then for such a branch, we reexecute the program together with a random modification on the input until the branch changes its decision, then say, the branch is resolved. Concretely, given a program $P$ (under the loaded binary code) and an input $m$ stored in $Addr_m = \{addr_0, addr_1, \dots\}$ as a finite set of addresses, the algorithm works in 2 consecutive phases:

1. Tainting: initialize empty lists: $T$ of the executed instructions, $Br(T)$ of the logged branches, and $ChkPt(T)$ of the saved checkpoints
   - Execute $P$ on $m$, before the execution of each instruction:
     - push it into $T$, and
     - if it is a branch then push it into $Br(T)$,
     - if it accesses (i.e. reads or writes) some elements of $Addr_m$ then save a checkpoint and push the checkpoint into $ChkPt(T)$.
   - When the execution finishes, for each $br \in Br(T)$ whose decision depends on values of some elements of $Addr_m$:
     - mark it unresolved, and
     - associate to it the nearest checkpoint $chkpt(br) \in ChkPt(T)$.
2. Rollbacking:
   - Rollback $P$ to the first checkpoint in $ChkPt(T)$,
   - For each $br \in Br(T)$ which is still marked as unresolved:
     - step 1: extract its nearest checkpoint $chkpt((br)$,
     - step 2: modify randomly the parts of $m$ which affect $br$'s decision and rollback $P$ to $chkpt(br)$,
     - step 3: continue executing until meet $br$.
       - if $br$ is marked as resolved then continue executing to the next unresolved branch in $Br(T)$,
       - if $br$ is still marked as unresolved then: if the decision of $br$ changes then mark $br$ as resolved and rollback to $chkpt(br)$, otherwise go to the step 2.

*Note 3.* The rollbacking phase tries to resolve all branches marked as unresolved in the tainting phase: these branches depend on the initial input (as determined by the dynamic tainting). A subtle detail is that some branches which are not marked as unresolved may take new decisions in the rollbacking phase, that is because of the dynamic tainting is precise only for the execution on the initial input. But if they take new decisions then that means also they are resolved, so the algorithm marks them as *supplemental resolved branches*.

**Dynamic tainting** Called in the tainting phase, the function determines for each $br \in Br(T)$, which parts of $m$ affect $br$'s decision. It consists of 2 phases:

1. Tainting graph construction: before the execution of each $ins \in T$,
   - denote $c$ the execution order of $ins$,
   - draw a hyper-edge with label $(c, ins)$, its heads are read operands and tails are written operands of $ins$.

   The result of this phase is a hyper-graph $G$ called the tainting graph.
2. Dependency computation: for each node $opr$ of $G$ satisfying $opr \in Addr_m$,
   - construct $opr$'s dependent instruction list by collecting edge labels in a depth-first traversal (the traversal order is exactly the order $c$ in the edge's label) starting from $opr$, then
   - for each $br \in Br(T)$, the parts of $m$ affecting $br$, denoted by the set $mem\_dep(br)$, consists of $opr \in Addr_m$ so that $br$ is an element of $opr$'s dependent instruction list.

*Example 3.* The instruction locating at `0x3afe00ad25` with content `"mov eax, dword ptr [rsp+0x70]"` (extracted from `libresolv`) has its hyper-edge depicted in Figure 1. At running time, the operand `dword ptr [rsp+0x70]` points to the four addresses from `0x7fffa1a53940` to `0x7fffa1a53943`.

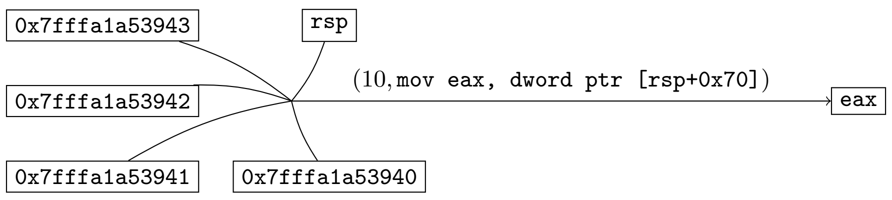


Fig. 1: Hyper-edge of (`0x3afe00ad25`, `"mov eax, dword ptr [rsp+0x70]"`)

**Nearest checkpoint computation** First we note that in the tainting phase, a new checkpoint is saved each time the executed instruction accesses some addresses in $Addr_m$. Moreover, in the step 2 (in the rollbacking phase of the branch resolving algorithm), the program $P$ is rollbacked to $chkpt(br)$ with random modification on certain parts of $m$, so this rollback is required to be coherent. In other words, the checkpoint $chkpt(br)$ must satisfy: *the random modification on these parts of $m$ does not conflict with the execution before $chkpt(br)$*.

Here we chose a simple solution: $chkpt(br)$ is the first element of $ChkPt(T)$ satisfying its next executed instruction reads some addresses in $mem\_dep(br)$. To see why $chkpt(br)$ with the random modification on $mem\_dep(br)$ constitute a coherent rollback, we observe that the execution before $chkpt(br)$ does not access any address of $mem\_dep(br)$ and then the execution is independent from $mem\_dep(br)$, in other words any modification on data at addresses of $mem\_dep(br)$ does not lead to conflict with the execution before $chkpt(br)$.

**Input synthesis** To resolve a branch, we modify randomly the part of the input delineated by the tainting analysis.

## 4 Practical implementation and result

The practical part of the paper is realized with the help of the Pin Dynamic Binary Instrumentation framework [3]. For a very short introduction, Pin consists of a *JIT compiler* which transforms original codes of the examined program into instrumented codes, and a *dispatcher* which then executes instrumented codes. By writing an appropriate *Pintool*, the analyzer controls this code transformation and then obtains loading and running time information of the program, as well as modifying its execution.

### Implementation

A technical problem in implementing our proof-of-concept is the *rollback* mechanism which is not supported natively in Pin. The closest feature is the manipulation on the CONTEXT, a data structure containing the state of the processor, a similar feature on the program's memory space is not available.

In previous approaches [6, 9], the rollbacks are implemented by a system emulator such as QEMU which allows saving/restoring the memory state by taking snapshots. We implement another light-weight mechanism that logs only the state modification when some instruction writes in memory:

- Checkpoint saving:
  - Save the state of the processor using Pin's API,
  - Initialize an empty logging set $L$, that will consist of pairs $(addr, val)$.
- Checkpoint logging (execute before the execution of a memory-written instruction *ins*): for each written address *addr* of *ins*,
  - construct a pair $(addr, val)$ where *val* is the value currently stored (i.e. before being overwritten) at *addr*, and
  - if *addr* does not exist in any element of $L$ then add this pair into $L$.
- Checkpoint restoring:
  - Restore the state of the processor using Pin's API,
  - For each element in the logging set, restore the value at the address *addr* of the element by $orig\_val$.

The disadvantage of this mechanism is that it does not allow switching directly between checkpoints, but it is sufficient for the branch resolving algorithm.

### Experimentation

*Tested programs and inputs* The Pintool has been tested for programs: `ping`, `wget`, `aria2c`. The input for each tested program is marked as the first message received from the network. To perform that, the system call `recvfrom` is intercepted by tracing the exception `syscall` with $\texttt{id} = 45$ (stored in `rax`).

*Testing parameters* The main parameters are the *size of the linear trace* (i.e. number of executed instructions), and the *bounded value of total rollbacks* (abbr. BVTR). An auxiliary parameter is the *bounded value of local rollback* (abbr. BVLR): that is, for some given branch, the Pintool will stop to rollback on it once the number of rollbacks reached the BVLR.

*Testing results* At the first execution with a normal input, the *number of logged branches* is determined by the number of branches depending on the input. The rollback executions later try to resolve these logged branches, and the result is stored in the *number of resolved branches*. The *resolved rate* is determined by the ratio between the number resolved branches and of the logged branches. An auxiliary result is the *number of actual used rollbacks*.

*Modulation* The program may receive a different normal input in each normal execution, and the algorithm changes randomly the input. To increase the reliability of the experiment, the program is executed under the instrumentation with the same parameter triplet (i.e. the size of the linear trace, the bounded value of total rollbacks, and the bounded value of local rollback) multiple times. For each time, the above results are logged; and the final results are determined by the mean value of logged results.

## Results

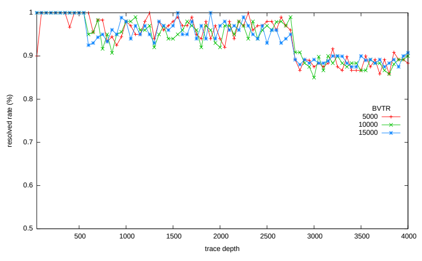
Following is the result obtained in testing `wget` 50 times, some quantitative information is shown in Figures 2a and 2b. For both of them, the results seem to be quite predictable: the resolved rate increases when the BVTR increases. Observe that it decreases when the size of the linear trace increases: that is because of the number of branches to be resolved increases with the length of trace. A dropout of the rate at some value around 2750 of the linear trace suggests that there exist some very hard to resolve branches occurring around this value. Indeed, the branch and its affecting instruction list is at $2773, 2772$:

```
2772  0x3afe00b9d8   cmp word ptr [r11+0x6], 0x0
2773  0x3afe00b9de   jnz 0x3afe00ba30
```

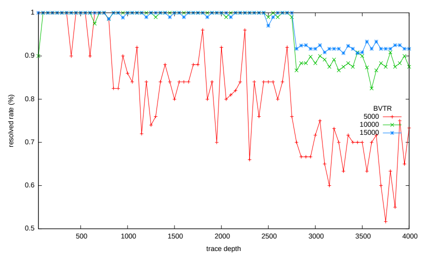
The algorithm has detected that to resolve the branch at 2773, it is enough to go back to the instruction at 2772. Though the size of this list is 2, the chance of success is $2^{-16}$ only.

Moreover, with forms consisting of two separated constant ordinate parts, the curves actually imply that the decrease of resolved rate is mostly because of the hard branches, barely by the size of the linear trace and by the bounded value of rollbacks. That is shown also in Figures 2c and 2d: the numbers of actual used rollbacks are much smaller than their bounded values, that signifies beside the hard ones, the branches are resolved after a small number of rollbacks only.

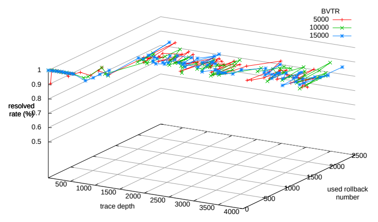
The difference between curves in Figures 2a and 2b is explained by the granularity of the BVLR with respect to the BVTR. If the BVTR is sufficiently large



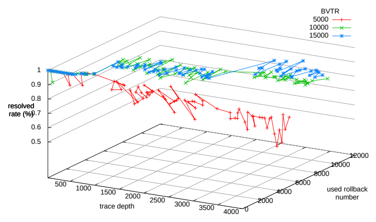
(a) BVLR = 500



(b) BVLR = 5000



(c) BVLR = 500



(d) BVLR = 5000

Fig. 2

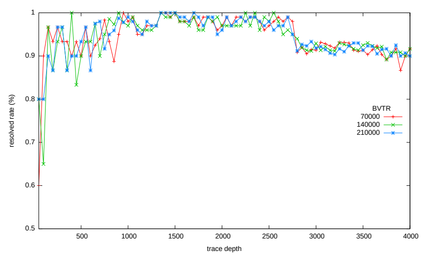
(e.g. the cases of 10000 and 15000), a higher BVLR increases the resolved rate because each branch is better tested. But if the BVTR is small (e.g. the case of 5000) a higher BVLR decreases the resolved rate because some hard branch will consume the most part of the BVTR, that leaves the other branches unresolved.

## 5 Discussion about improvement

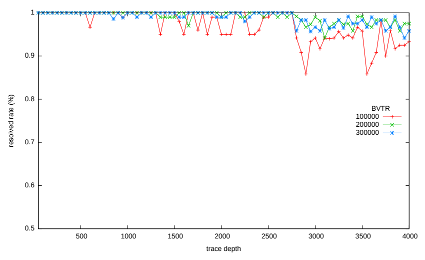
It must be stressed that the branch resolving algorithm is a kind of fuzzy testing (its actual improvement is to reduce the number of the executed instructions in tests). Though showing impressive results for the tested programs, it by no mean solves the radical problem of the testing approaches [10]. This fact is indeed shown in experimental results: some branch's decisions are extremely hard to take by just fuzzy testing.

A naive improvement is to increase the BVLR while keep its granularity (as in Figures 3a and 3b), but that makes the number of actual used rollbacks increase exponentially (as in Figures 3c and 3d): we have to pay about 100000 rollbacks to increase the resolved rate to 98% in comparison with less that 2500 rollbacks for the resolved rate at 90%.

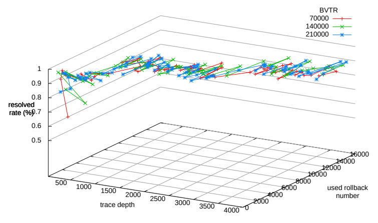
However, our approach is orthogonal to ones in the vein of *the dynamic test generation* [2, 4, 10] which are applied as main techniques of researches on malware's code exploration [6, 9]. Naturally, these techniques can be used to strengthen our algorithms.



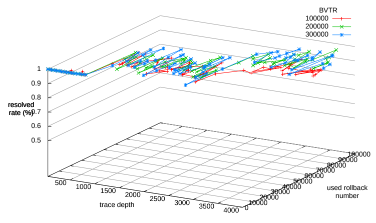
(a) BVLR = 7000



(b) BVLR = 70000



(c) BVLR = 7000



(d) BVLR = 70000

Fig. 3

## References

[1] J. C. King. “Symbolic Execution and Program Testing”. In: *CACM* 19.7 (1976), pp. 385–394.

[2] P. Godefroid et al. “DART: Directed Automated Random Testing”. In: PLDI. 2005, pp. 213–223.

[3] C.-K. Luk et al. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. In: PLDI. 2005, pp. 190–200.

[4] C. Cadar et al. “EXE: Automatically Generating Inputs of Death”. In: CCS. 2006, pp. 322–335.

[5] J. Caballero et al. “Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis”. In: CCS. 2007, pp. 523–529.

[6] A. Moser et al. “Exploring Multiple Execution Paths for Malware Analysis”. In: SSP. 2007, pp. 231–245.

[7] A. Moser et al. “Limits of Static Analysis for Malware Detection”. In: ACSAC. 2007, pp. 421–430.

[8] M. D. Preda. “Code Obfuscation and Malware Detection by Abstract Interpretation”. PhD thesis. 2007.

[9] D. Brumley et al. “Automatically Identifying Trigger-based Behavior in Malware”. In: *Botnet Analysis and Defense*. 2008, pp. 65–88.

[10] P. Godefroid et al. “Automated Whitebox Fuzz Testing”. In: NDSS. 2008, pp. 151–166.

[11] J. Caballero et al. “Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-Engineering”. In: CCS. 2009, pp. 621–634.

[12] P. M. Comparetti et al. “Prospex: Protocol Specification Extraction”. In: SSP. 2009, pp. 110–125.

[13] M. Egele et al. “A Survey On Automated Dynamic Malware-Analysis Techniques and Tools”. In: *ACM Computing Survey* 44.2 (2012).

## A Appendix

### Nearest checkpoint calculation

```
Input: A list $chkpt\_list$ of checkpoints and a set $mem\_deps$ of pairs $(b, m_b)$.
Output: A set $branch\_min\_checkpoint$ of pairs $(b, chkpt)$ where $b$ is a branch
        and $chkpt$ is $b$'s nearest checkpoint.
begin
    $branch\_min\_checkpoint = \emptyset$;
    foreach $(b, m_b) \in mem\_deps$ do
        foreach $chkpt \in chkpt\_list$ do
            if $accessed\_mem\_addrs\,(chkpt) \cap m_b \neq \emptyset$ then
                break;
            end
        end
        $branch\_min\_checkpoint \leftarrow branch\_min\_checkpoint \cup \{(b, chkpt)\}$;
    end
end
```

**Algorithm 1:** Nearest checkpoint calculating

## Dynamic tainting

**Input**: A triplet $(P, m, l)$ consisting respectively of the program, its input and the maximum length of the examined trace.
**Output**: A set *mem_deps* of pairs $(b, m_b)$ where $b$ is a branch and $m_b$ is a set of memory parts of $m$ which affect to $b$'s decision.

```
begin
    $c = 0$;
    $G = \emptyset$ ;                                          /* taint graph */
    $ins\_list = \emptyset$ ;                     /* list of executed instructions */
    /* Constructing $G$ along with executing instructions              */
    while $P$ does not halt and $c \leq l$ do
        $ins \leftarrow$ next instruction; $ins\_list \leftarrow ins\_list \cup ins$; $c \leftarrow c + 1$;
        $In_{ins} \leftarrow$ read operands of $ins$;
        $Out_{ins} \leftarrow$ written operands of $ins$;
        $Nodes\,(G) \leftarrow Nodes\,(G) \cup In_{ins} \cup Out_{ins}$;
        $label \leftarrow (c, ins)$;
        $Edges\,(G) \leftarrow Edges\,(G) \cup \{(i, j, label) \mid i \leftarrow In_{ins}, j \leftarrow Out_{ins}\}$;
        execute ins;
    end
    /* Constructing instruction dependency lists for $m$ ($m$ is stored
       in memory at a set of addresses $m = \{m_0, m_1, \dots\}$          */
    $dep\_ins\_list = \emptyset$;
    foreach $opr \in Nodes\,(G)$ do
        if $opr \in m$ then
            $ins\_list = \emptyset$;                  /* $opr$'s dependent instructions */
            $v = opr$;
            $lab \leftarrow min\_label\,(v\text{'s inedges})$;
            while $lab \leq min\_label\,(v\text{'s outedges})$ do
                $e \leftarrow min\_edge\,(v\text{'s outedges})$;  /* edge with minimum label */
                $ins\_list \leftarrow ins\_list \cup \{e\}$;
                $lab \leftarrow label\,(e)$;
                $v \leftarrow tail\,(e)$;
            end
            $dep\_ins\_lists \leftarrow dep\_ins\_lists \cup (opr, ins\_list)$;
        end
    end
    /* Constructing $mem\_deps$                                        */
    $mem\_deps = \emptyset$;
    foreach $ins \in ins\_list$ do
        if $ins$ is a branch then
            $dep\_mem\_list = \emptyset$;        /* $ins$'s affecting memory addresses */
            foreach $(opr, dep\_list) \in dep\_ins\_lists$ do
                if $ins \in dep\_list$ then
                    $dep\_mem\_list \leftarrow dep\_mem\_list \cup opr$;
                end
            end
            $mem\_deps \leftarrow mem\_deps \cup (ins, dep\_mem\_list)$;
        end
    end
end
```

**Algorithm 2:** Dynamic tainting

## Branch resolving algorithm

```
Input: A quadruplet $(P, m, l, r)$ consisting respectively of the program, its
       input, the maximum length of trace, and the maximum number of
       rollbacks for each branch.
Output: A set $R$ of all resolved branches.
begin
    /* Tainting phase obtained the following                          */
    $T \leftarrow$ the execution trace of $P$ on $m$;
    $B(T) \leftarrow$ the list of branches on $T$;
    $CP(T) \leftarrow$ the list of checkpoints;
    $mem\_deps(T) \leftarrow$ the set obtained by dynamic tainting;

    /* Rollbacking phase                                              */
    $B_m(T) = \{b \mid b \leftarrow B(T), m_b \neq \emptyset\}$;
    while the last element of $B_m(T)$ is not resolved or not passed do
        $b \leftarrow$ the first unresolved element in $B(T)$;
        $m_b \leftarrow$ the parts of $m$ affecting $b$'s decision;
        $c_b \leftarrow b$'s nearest checkpoint;
        $r_b \leftarrow 0$ ;                              /* rollback counter */
        while $c_b \neq 0$ do
            modify randomly $m_b$; rollback to $c_b$;
            foreach $ins \leftarrow$ next instruction do              /* $ins \in T$ */
                if $ins \in B(T)$ then        /* $ins$ is a conditional branch */
                    if the target of $ins$ is not the one in $T$ then
                        mark $ins$ as resolved; $R \leftarrow R \cup \{ins\}$;
                        if $r_b \geq r$ then           /* counter reaches limit */
                            $c = c_b$; $c_b = 0$; restore $m_b$; $clean\_rollback(c)$;
                        else
                            /* fuzz testing will lost $T$ if continues,
                               then goes back                              */
                            $r_b \leftarrow r_b + 1$; modify randomly $m_b$; $rollback(c_b)$;
                        end
                    else
                        if $ins == b$ then
                            if $r_b \geq r$ then       /* counter reaches limit */
                                mark $ins$ as passed;
                                $c = c_b$; $c_b = 0$; restore $m_b$; $clean\_rollback(c)$;
                            else
                                /* backs again to try resolving $b$       */
                                $r_b \leftarrow r_b + 1$; modify randomly $m_b$; $rollback(c_b)$;
                            end
                        end
                    end
                end
                execute $ins$;
            end
        end
    end
end
```

**Algorithm 3:** Branch resolving