Efficient Extraction of Malware Signatures Through System Calls and Symbolic Execution: An Experience Report

Eduard Baranov, Fabrizio Biondi, Olivier Decourbe, Thomas Given-Wilson, Axel Legay, Cassius Puodzius, Jean Quilbeuf, Stefano Sebastio

Efficient Extraction of Malware Signatures Through System Calls and Symbolic Execution: An Experience Report

Eduard Baranov
Inria Rennes

Fabrizio Biondi
Inria Rennes

Olivier Decourbe
Inria Rennes

Thomas Given-Wilson
Inria Rennes

Axel Legay
UCL Louvain

Cassius Puodzius
Inria Rennes

Jean Quilbeuf
Inria Rennes

Stefano Sebastio
Inria Rennes

ABSTRACT

The ramping up use of network connected devices is providing hackers more incentives and opportunities to design and spread new security threats. Usually, malware analysts employ a mix of automated tools and human expertise to study the behavior of suspicious binaries and design suitable countermeasures. The analysis techniques adopted by automated tools include symbolic execution. Symbolic execution envisages the exploration of all the possible execution paths of the binary without neither concretizing the values of the variables nor dynamically executing the code (i.e., the binary is analyzed statically). Instead, all the values are represented symbolically. Progressing in the code exploration, constraints on symbolic variables are built and system calls tracked. A satisfiability-modulo-theory (SMT) checker is in charge of verifying the satisfiability of the collected symbolic constraints and thus the validity of an execution path. Unfortunately, while widely considered promising, this approach suffers from high resource consumption. Therefore, optimizing the constraint solver and tuning the features controlling symbolic execution is of fundamental importance to effectively adopting the technique. In this article, we identify the metrics characterizing the quality of binary signatures expressed as system call dependency graphs extracted from a malware database. Then, we pinpoint some optimizations allowing to extract better binary signatures and thus to outperform the vanilla version of symbolic analysis tools in terms of malware classification and exploitation of the available resources.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; • **General and reference → Empirical studies**; *Evaluation*; • **Software and its engineering →** *Constraint and logic languages*;

KEYWORDS

Malware analysis, Symbolic execution, Empirical studies, System call dependency graph, Constraint Programming, Satisfiability Modulo Theories (SMT)

CODASPY 2019, March 2019, Dallas, Texas USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Eduard Baranov, Fabrizio Biondi, Olivier Decourbe, Thomas Given-Wilson, Axel Legay, Cassius Puodzius, Jean Quilbeuf, and Stefano Sebastio. 2019. Efficient Extraction of Malware Signatures Through System Calls and Symbolic Execution: An Experience Report. In *Proceedings of ACM Conference on Data and Application Security and Privacy (CODASPY 2019).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Due to the intrinsic vulnerability of software, security threats for network connected devices are common news. Also, carelessness and lack of user education allow an easy diffusion of malicious software (*malware*), faster than the ability of security analysts to study threats and provide countermeasures. Nowadays security analysts, even if supported by analysis tools, still need to strongly rely on their experience and ability to manually analyze malware. Therefore, enhancing the capabilities of automated tools assumes a central role in dealing with novel software threats.

The analysis of malware behavior is usually divided in *dynamic* and *static*. In dynamic analysis [26] malware are executed in an isolated and virtualized environment to understand how they interact with the operating system [52]. Unfortunately, the report generated by observing malware actions is dependent upon the context emulated by the virtualized environment. This scenario leads to two drawbacks: it is not always possible to know the contextual conditions triggering malicious behavior, and techniques to hinder dynamic analysis such as sandbox detection are a common practice nowadays [12]. Static techniques [5] instead perform the analysis of possible execution paths by exploring the code without executing it on a real machine and thus conceivably providing a wider knowledge of the malware's behavior. One of the hurdles of static analysis lies in the absence of the malware source code, requiring the use of a disassembler. The action of the disassembler can thus be hindered by the malware with the use of obfuscation techniques [6, 33] such as virtualization [18, 19, 46, 50], Just-In-Time compilation [9, 18, 60], polymorphism [8, 39, 54] and packing [31, 36, 58].

Static code analysis based on the assembly-like code extracted by the disassembler can be performed *concretely* or *symbolically*. In the concrete analysis, the execution trace over the disassembled code is steered by a provided input. The ability to explore the behavior of the binary is thus limited by the chosen input. Therefore, it could potentially never explore the whole *Control Flow Graph (CFG)* (i.e., the graph representing the set of all traversable paths during the program's execution) and thus never unveil a potentially malicious

behavior. To overcome this limitation, symbolic execution traverses the code considering symbolic input variables in place of concrete ones. In this way, all the possible execution paths are taken into account and the CFG can be potentially fully explored. To achieve such a goal, at each execution step all the *constraints* added to the symbolic variables during the symbolic execution must be considered. In case of execution branches such as conditional jumps (e.g., `if`, `while` or `for` statements) the whole program state is replicated for each new branch and the corresponding jump conditions are added to the symbolic variables in the form of constraints. It is thus crucial to efficiently verify the feasibility of the collected constrained expressions to determine whether a given execution path is feasible or not, i.e corresponds to a trace that can actually be executed. These *Satisfiability Modulo Theories* (SMT) decision problems are solved with the aid of a *theorem prover* such as the Microsoft Research's *z3* [22]. We refer the interested reader to [5] for an exhaustive survey on symbolic execution techniques. It is worth to point out that malware analysts employ a mix of tools during their analysis since, given complexity and heterogeneity of the current malware families, no "silver bullet" exists.

To perform its malicious actions, malware interact with their host system through *System Calls (SCs)* [2]. SCs represent the way in which a software requests any service to the kernel of the operating system. The available services span from process, memory, and files management, to device and network handling. The System Call Dependency Graph (SCDG) built from the logical dependency of such SCs (i.e., the information flow propagated through these SCs), originally proposed in [17, 27], has been proved [13, 34, 47] to be a good data structure to represent a behavioral signature of a malware sample, and thus to classify malware from cleanware. E.g., [10] detects with a high accuracy binary samples of the Mirai malware [3]; Dam and Touili [20] outperformed widely-used anti-viruses by detecting malware samples that were otherwise unidentified; and Dimjašević et al. [24] used system call dependencies extracted through a dynamic sandboxed execution to detect Android malware. Unfortunately, even a trivial obfuscation technique could request a high number of irrelevant SCs (e.g., just by opening and closing files in a loop, adding spurious calls or changing the order of independent calls) with the aim of producing a different graph. For this reason, building effective SCDGs for malware classification is not an easy task, requiring care both in malware sample exploration and in graph construction and classification.

Contributions. The goal of this paper is to discover the properties defining good binary signatures based on SCDGs and to determine how to extract such signatures efficiently using symbolic execution. The contributions of the paper are:

- identify quality characteristics and evaluation metrics of binary signatures based on SCDGs (and consequently the key properties of the execution traces), that characterize signatures able to provide high-precision malware classification;
- study the impact of the configuration of the SMT solver and symbolic execution framework, and understand their interdependencies with the aim of efficiently extracting SCDGs in accordance with the identified quality metrics.

A design of experiments approach allowed to infer how heuristics adopted during both the symbolic execution and SCDG building

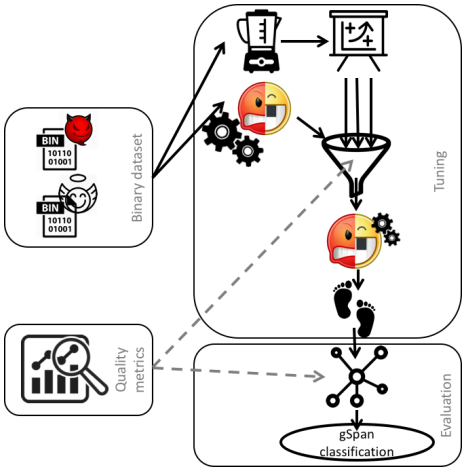


Figure 1: Our workflow to efficiently extract SCDG-based signatures through symbolic execution.

processes affect the quality of the malware classification. The evaluation, over a dataset of 225 malware and cleanware, proves our ability to extract effective binary signatures.

Our approach is summarized by the workflow represented in Figure 1. Symbolic executions have been performed over a dataset composed by malware and cleanware, extracting execution traces. Such traces constituted the input information to built the SCDGs, which have later been mined by the gSpan [59] common subgraph algorithm to build binary signatures. Then, the quality of the signatures has been evaluated according to their ability to discern malware from cleanware, and to classify malware according to their family. Successively, we explored heuristics of the SMT solver (extracting symbolic constrained expressions in SMT-LIB v2 format, Section 4) and the symbolic execution framework itself. Finally, efficacy of the heuristics, their interactions and the graph metrics characterizing good extractions have been evaluated (see Section 6).

Both learning and evaluation processes have been carried out through a *5-fold cross-validation* to reduce the chance of overfitting. As a general reference to the statistical concepts used hereinafter we refer to [1]. Experiments have been performed on a cluster constituted by 12 machines having four Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00 GHz with 132 GB of RAM each, running Ubuntu 16.04 LTS (xenial, kernel version: 4.4.0-21-generic), Python 2.7.14, Angr build 7.8.8.1 and z3 4.5.1.0.post2. The binary dataset is composed by a total of 225 samples, i.e., 25 cleanware and 25 binaries for each of 8 malware families (namely, *couponmarvel*, *gamemodding*, *installbrain*, *multiplug*, *jeefo*, *detroie*, *mira* and *addrop*) belonging to various malware types (according to the anti-viruses aggregator VirusTotal) such as: trojan, adware, PUP (Potentially Unwanted Program), downloader and virus.

Paper Structure. The rest of the paper is organized as follows. SCDGs building heuristics, graph metrics and performance indicators for a SCDG-based malware classifier are described in Section 2. Section 3 provides a general introduction on the binary analysis framework *Angr* [45] considered in our experiments, highlights limitations of symbolic analysis, and how to address them. Details

on the use of the SMT solver in Angr and how we improved its performance for malware analysis are discussed in Section 4. Section 5 presents our experimental methodology whereas results, analysis, threats to validity and discussion over the conducted experimental campaign to understand the impact of the features controlling the symbolic execution are introduced in Section 6. Finally, Section 7 concludes the paper providing some final remarks and outlining possible future research efforts.

2 MALWARE CLASSIFICATION VIA SCDG

As discussed in Section 1, SCDGs have been proved as a good approach to build binary behavioral signatures. Thanks to a common subgraph mining algorithm such as gSpan [59], it is possible to evaluate the similarity among the SCDG representing the behavior of a new sample and ones in the training set. SCDGs built from the binaries in the training set constitute the samples of the supervised learning phase of the classifier i.e., graphs are labeled as extracted from malware or cleanware. Intuitively, an SCDG has high quality from the malware classification point of view if using it leads to high-detection, low-false-positive malware classification.

In this section, we describe different heuristics to build SCDGs from execution traces obtained by the symbolic analysis and briefly mention some of the graph metrics later sifted through as possible indicators of graph quality (see Section 6). Then, we present how the SCDG-based malware classification works and how the classifier is evaluated.

2.1 SCDG Building Heuristics

Execution traces consist of sequences of system calls alongside their address and arguments. Address, order of execution and arguments allow one to understand the relations between the calls and thus to build a directed graph. In SCDGs, vertices represent the requested system calls and edges the information flows between them (i.e., one of the input or output arguments of a call is the input of another call). Labeled vertices allow one to identify the service requested to the operating system. In case of unrelated calls to the same service, different vertices are created. Since not all the system calls are related, from each execution trace a graph with several connected components is built.

Moreover, since symbolic execution considers all the possible paths of a given binary, several execution traces could be generated corresponding to the conditional branches that have been evaluated satisfiable by the SMT solver. For efficiency reasons, before actually building the SCDGs, all the traces constituting the prefix of another one are discarded. Finally, gSpan is used to find common subgraphs (see Section 2.2), as proposed by Palahan et al. [38].

From the execution traces, SCDGs could be built in different ways. In particular, we consider three parametric heuristics:

- *merge-calls*: whether system calls having same name and address must be merged or not
- *disjoint union / traces-merge*: whether the different graphs built from the same binary must be considered in a disjoint union or a single graph must be obtained merging the traces having a common prefix (similarly to the system call dependency trees proposed in [35])
- *min-trace-size*: does a minimum number of calls have to be present in a trace for it to be valid i.e., shorter traces are discarded.

Based on how these parameters are instantiated, different SCDGs could possibly be obtained from the same set of traces. By providing different sets of SCDGs in input to the graph mining phase (as described in Section 2.2), various classifiers could be built. According to our past experience and by observing the length of the extracted traces, we tested the *min-trace-size* instantiated to: 0 (i.e., do not place any restriction) and 10 (see Section 5 and 6).

Heuristics used by the symbolic execution framework affect quality and quantity of the collected information (contained in the traces), whereas the SCDG building and mining processes determine how to use such information to build a behavioral signature.

From the set of graphs built from each binary the following metrics have been computed: number of unique calls (i.e., calls to the same system call from the same address of the binary address space), number of edges, number of nodes, max size of the weakly connected component (i.e., maximal subgraph in a directed graph for which it exists a path between each couple of nodes), number of weakly connected component, graph density (defined as: $\frac{\#edges}{\#vertex \cdot (\#vertex-1)}$), and number of unique edges (i.e., information flow having same source, destination and arguments).

2.2 SCDG Classification and Evaluation

Given a set of graphs and a *minimum support threshold* (i.e., the minimum percentage of graphs in the dataset containing a given subgraph to consider the graph relevant), the gSpan mining algorithm finds all the common subgraphs between the graphs in the set. Having SCDGs labeled as extracted by malware or cleanware, as a matter of fact, *malware semantic signatures* are composed by the n largest sub-graphs in the set of SCDGs extracted from malware binaries. In the interest of time, in our experiments we considered only the largest sub-graph (i.e., $n = 1$).

To classify a new binary sample by its SCDG(s), gSpan is called to find subgraphs that were present in the signatures of the learning dataset. The *similarity metric* between two graphs is defined considering the percentage of the number of edges in the malware semantic signature included in the SCDG of the new sample to classify (hereinafter *similarity threshold*).

For a binary classifier (i.e., discerning malware from cleanware in our context), it is possible to derive several metrics: (i) *recall* (sometimes referred also as *sensitivity*) as the number of correct positive classifications over the total number of positive cases, (ii) *precision* as true positives over all the positive results returned by the classifier, and (iii) *accuracy* (or *trueness*) as rate of correct classifications over the total number of samples. A simple but effective metric to measure the accuracy of a binary classification is represented by the *F-score* (also referred as F_1*-score*). It is computed as the harmonic mean of precision and recall: $2 \cdot \frac{precision \cdot recall}{precision + recall}$. By construction, the closer to 1 is the F-score the better it is. Since in our context we are also interested in classifying the malware sample's family, we adopted the *micro-average F-score* which foresees the count of correct and wrong classifications (true positives, false positives, and false negatives) for each class independently before appropriately summing them up for computing precision and recall.

3 SYMBOLIC EXECUTION WITH ANGR

In this section Angr [45], the binary analysis framework adopted in our experiments, is introduced with the purpose of illustrating how symbolic execution works in practice and the central role covered by the SMT solver (z3 in the case of Angr). Moreover some limitations and challenges (and possible countermeasures) of symbolic execution are mentioned.

3.1 From Binaries to Symbolic Execution

Malware often spreads in the form of executable files. In case of static symbolic analysis, the first step performed by the security analysts concerns the recovery of the corresponding assembly-like instructions by means of disassembler tools on which the static symbolic code execution is performed. Angr is a Python framework that allows one to perform both disassembly and symbolic execution. In more detail, its main modules are:

- `CLE`: is in charge of loading a binary alongside any dependent library, and mapping these to a single program's memory space;
- `Capstone`: is a multi-architecture disassembler, thus transforms the binary into the corresponding assembly code;
- `VEX`: provides an *Intermediate Representation (IR)* of the assembly code abstracting away differences among architectures (e.g., ARM, AMD64, x86) in terms of registry name, memory access, segmentation, etc.
- `Claripy`: is an interface to the SMT solver z3;
- `Execution Engine`: performs the static execution (concrete or symbolic) over the VEX IR by calling Claripy for each operation involving new constraints over symbolic variables.

In this work, we are mainly focused on three components: the VEX IR, the SMT solver wrapper Claripy, and the Execution Engine. It is worth remarking that the execution is not on the source code specified in a high-level programming language (since it can not be easily reconstructed from the binary) but on the VEX IR where each variable, constant or registry is represented as a *bitvector* (vector of bits).

The applicability of the symbolic analysis of Angr in analyzing real threat has been proved e.g. in [4] while studying the behavior of a remote access trojan. Even if in this work we used Angr and performed an in-depth experimental evaluation of its heuristics, other symbolic execution frameworks such as Mayhem [14], S^2E [16] and Triton [42] provide similar heuristics or, in case of lack of the corresponding heuristic, application programming interfaces to integrate it.

3.2 Calling the SMT Solver

During symbolic executions, new constraints over symbolic variables are potentially added at each step of the execution and in particular in the case of conditional jumps on the VEX IR code. As a matter of fact, these cases require the clone of the current state for each branch of the execution. For efficiency reasons, it is worth understanding if all of the execution traces from these new branches are really feasible or not according to the past execution. Continuing the execution towards a path that is unfeasible based on the collected constraints is a waste of both time and memory resources. To avoid this, the Execution Engine calls the SMT solver and prunes any potential unfeasible execution path. The z3 solver answers to SMT decision problems with `sat`, `unsat` or `unknown`. The first two answers clearly mean that the *Constraint Satisfaction Problem (CSP)* has solution (satisfiable) or not (unsatisfiable), whereas `unknown` means that the solver is not able to find a solution e.g., because the problem is too hard to be solved with a given algorithm and a timeout has been reached.

In particular, the constraint solver is called through Claripy in the following functions:

- *satisfiable()*: checks if the constraints assigned to the current state are satisfiable or not
- *batch_eval()*: evaluates a list of constrained expressions and provides solutions that satisfy them jointly
- *min() / max()*: respectively minimum and maximum value compatible with the constraints on a symbolic expression
- *is_false() / is_true()*: return a boolean corresponding to the feasibility of a given expression
- *simplify()*: tries to simplify a constrained expression in a way that could make subsequent satisfiability check easier.

3.3 Limitations of Symbolic Execution

As highlighted in the introduction (Section 1), the symbolic execution is a very powerful tool to understand the behavior of a binary. However, it is worth to mention also some of its limitations such as:

- interaction with the environment: in case of system calls or external (dynamic) libraries, appropriate *models* need to be defined because the binary does not contain them.
- loops: are a critical point since, as already discussed, for each conditional branch the state must be cloned and the execution trace split in two. Loops thus exacerbate the consumed resources. Detecting the loop and stopping to unwind it upon reaching a given threshold could constitute a feasible countermeasure. In such a case the loop exploration will be halted (stashing the corresponding state) and priority will be given to the other active states. We propose and evaluate a novel computationally lightweight loop detection for conditional jumps spawning new execution branches in Section 6.
- memory consumption and state space explosion: both are related to the functioning of the symbolic execution itself where, unlike in concrete executions, the least amount of assumptions are done. In case of such problems, other than trying to optimize the SMT solver (like in this paper) the sole possible approach is to pour all the additional knowledge available for the binary under analysis. This could be acquired through a disassembler [25, 40]. E.g., restriction on the bits for the variables in input could be represented as additional constraints to the symbolic variables. Another viable strategy is *concolic* execution (a crasis of concrete and symbolic) where symbolic variables are used only when strictly needed and concrete values are assigned to the others [44, 56].

It is worth noting that in general symbolic execution faces an *accuracy/performance trade-off*. More resources potentially lead to better code exploration. The optimization to the core components of the symbolic execution (e.g., the SMT solver and the exploration

Table 1: Composition of our benchmark of SMT expressions extracted through symbolic executions (having a timeout of 2 mins per binary).

	Dataset: Malware + Cleanware
# of binaries	906
extraction from `simplify()`	54460
extraction from `satisfiable()`	103881
extraction from `batch_eval()`	47585
extraction from `max()`	177439
extraction from `min()`	112712

heuristics, exploited respectively in Section 4 and 5) has thus the effect of reducing the resources consumption with benefits reflected in the whole binary analysis process.

4 SMT SOLVER OPTIMIZATION

Angr's solver engine Claripy provides an interface with the backend SMT solver. Currently the sole supported solver is z3 [22] but other solvers can be supported by writing an appropriate backend.

Our approach to optimize the SMT solver performing symbolic execution for malware analysis is depicted in the *tuning* block of Figure 1. First, a benchmark of SMT expressions extracted from symbolic executions of binaries (both malware and cleanware) has been built. Then the different simplification / solver strategies provided by the z3 (see the following Section 4.1) have been evaluated on them, identifying the most promising heuristics (see Section 4.2). Finally, Angr has been engineered to use the appropriate heuristic for each call to z3. The performance evaluation of this customized version of Angr is presented and discussed in Section 6.

To avoid any potential overfitting of z3 with the characteristics of the dataset used in the evaluation (see Section 1), here we considered a larger datasets constructed with samples not included in the former.

The extracted expressions are formatted according to the SMT-LIB v2 format [7]. This format aims at promoting a common input and output language for SMT solvers in order to ease the exchange of benchmarks among the research community. The z3 heuristics have been evaluated over this benchmark considering both performance (i.e., execution time) and accuracy (i.e., number of `sat` / `unsat` answers instead of `unknown`).

A summary of the composition of our benchmark built extracting SMT expressions from the symbolic execution of binaries is presented in Table 1.

4.1 Tuning the Z3 Solver

The z3 solver is equipped with a few hundred parameters grouped in modules. These parameters include both Boolean and numeric parameters (there are also some string parameters corresponding to predefined option sets). The purpose of such parameters is to tune z3 with the aim of speeding up the evaluation of the constraints in case of any a-priori knowledge of its features.

By using the more than 300 available parameters, the Microsoft Research team working on z3 has identified about 100 known problems and designed optimized approaches (*tactics* in the z3 jargon) to tackle them. E.g., there are built-in tactics for closed quantifier-free formulas over the theory of fixed-size bitvectors (`QF_BV`), non-linear integer arithmetic with uninterpreted sort and function symbols (`UFNIA`) or closed linear formulas over linear integer arithmetic (`LIA`). In general, even knowing the class of problems to which an expression belongs to and having a tactic tailored for that, no benefit is certain. As a matter of fact, tactics are just heuristic strategies. Their approach in solving the constraints associated to the SMT expression is not guaranteed to be optimal in terms of computational time or ability to find a solution.

Such heuristics tend to be highly tuned for known classes of problems but generally perform badly for new classes [23]. Brand new tactics can be easily defined from scratch considering the available parameters. Tactics can even be combined in several ways. A total of eight combinators of tactics (*tacticals* [23]) are available. The bottom-line idea is to apply the provided list of tactics in: sequence, parallel or alternative in case of failure of the previous ones on the list. E.g., to simplify the constraints, Angr uses a customized tactic constituted by the sequential application of five tactics provided by z3 (i.e., `simplify`, `propagate-ineqs`, `propagate-values`, `unit-subsume-simplify` and `aig`). These respectively correspond to: simplification, removal of the inequalities, constant propagation and simplification of set of clauses including a unit clause (i.e., the one composed by a single boolean literal).

If no tactic is provided (named `NO.TACTIC` hereinafter), the z3 solver will try to infer the logic to which the expression belongs to (using *probes*) and to apply the corresponding tactic (if available). Otherwise, the `smt` tactic is executed [1].

Previous works have considered the optimization of the solver for a specific class of formulas e.g., quantified bitvector [53] or linear integer / real arithmetics [28]. Instead, in our work we have not any a-priori knowledge about the expressions generated during the symbolic execution and the different interactions with the solver (see Section 3.2). We just expected that, even if the different binaries would have shown a quite heterogeneous set of SMT expressions, their logic would have been generally traced back to the presence of bitvectors.

4.2 Expressions evaluation with z3

We built a benchmark constituted by all the SMT expressions produced by analyzing with Angr a dataset constituted by both malware and cleanware (see Section 4). The performance of all the z3 built-in tactics (plus `NO.TACTIC`) in solving the benchmark has been evaluated. For some of such heuristics, the time required to solve some expressions was prohibitive, up to several hours without finding a definitive answer. Therefore, from our past experience considering a reasonable time to analyze a single binary and the number of calls to the solver, a timeout of one minute for solving each expression has been adopted before forcing the heuristic to give up, answering `unknown`.

[1] The z3 source code in case no tactic is provided is available online at https://github.com/Z3Prover/z3/blob/master/src/tactic/portfolio/default_tactic.cpp.

Since Angr is developed in Python, evaluating the performance of tactics, among the several available bindings for various programming languages the `z3py` interface has been used. The evaluation of each heuristic against each SMT expression has been repeated 30 times measuring the execution time through the `timeit` Python module. To remove the dependency from any contingent spurious background process, mean and standard deviation have been considered in the following comparisons.

Angr's binary analysis platform faces a trade-off between resources consumption (in terms of both CPU time and memory) and ability to get a definitive answer about the satisfiability of a given SMT expression built during the symbolic analysis of VEX IR code. Having a resource budget, the more resources are spent by the solver on a given expression, the less will be left to further explore the binary. On the other hand, generally, more resources are required by the tactics able to provide a conclusive answer (i.e., `sat` or `unsat`) on the satisfiability check of an expression. In this latter case, the advantage lies in the ability of pruning an unreachable execution path found by the symbolic analysis and in general steering the binary exploration towards more interesting portions of the code. Therefore, the solver, instrumented with a tactic having a greater accuracy (i.e., able to provide more definitive answers), will pay back for the higher resource cost only after the satisfiability check of an expression has been concluded. If the solver exceeds the resource budget, Angr execution is halted. Two metrics have been considered in evaluating the effectiveness of a tactic: execution time and ability to successfully perform a satisfiability check. As baseline tactic, we have considered the `NO.TACTIC`. A summary of the performance of the selected tactics over the benchmark of SMT expressions is presented in Table 2.

Since heuristic proof strategies are rarely "one size fits all" [23] in the remain of this section, to motivate the use of a parallel tactical with the selected tactics, through scatter plots we compare the default tactic versus the chosen one for each call of Claripy to z3.

The best tactic on the expressions extracted from `simplify()` is the `qfufbv_ackr`. This tactic corresponds to the use of Quantified Free formulas (QF) over bitvectors (BV) with uninterpreted sort function (UF) and symbols solved with Ackermanization (i.e., replacing all the applications of UFs with fresh variables and adding constraints to enforce the functional consistency). Figure 2a compares `NO.TACTIC` and `qfufbv_ackr` on the execution time to solve the SMT expressions from `simplify()` in the benchmark. It is possible to note that with the exception of a very few binaries the benefit of using the `qfufbv_ackr` in place of the default tactic is clear.

Considering instead the expressions from `batch_eval()` the use of the `qflra` (QF linear real arithmetic i.e., Boolean combinations of linear polynomials over real variables) in place of the `NO.TACTIC` has a clear benefit over all the binaries but a very few (see Figure 2c).

Instead, as reported in the scatter plot in Figure 2b, none of the tactics proved to be consistently faster than the default one for the `satisfiable()` call (see the green line obtained by a simple linear interpolation model).

`max()` and `min()` account for most of the time spent by the solver (see the first row in Table 2). Selecting the right tactic, respectively `qfidl` (Boolean combinations of inequality constituted by differences between integers variables and constants) and `smt` (Boolean SAT-based SMT solver), the default configuration could be over-performed for all the binaries as shown in the scatter plots in Figure 2d and 2e.

It is worth to note that a tactic could fail to establish if a formula is satisfiable or not (e.g., if the formula uses integers whereas the tactic works and finds a solution over reals). Since a solution is found the exception is not thrown and such a tactic becomes the one to decide the outcome of the tactical and that the goal can not be reduced further even if it has returned as sub-goal the original one. In this case the trick consists in combining with an `And-Then` and `fail` tactic to signal a failure in case `non-true` or `non-false` subgoals are created. This trick has been used by us in case of the `solve-eqs` tactic when, thanks to a tactical, it has been combined with the others selected.

5 EXPERIMENTAL METHOD

In this section the considered symbolic execution heuristics and our experimental method are described.

5.1 Symbolic Execution Heuristics

To limit resource consumption, a set of parameters is usually available in every symbolic execution tool:

- `timeout`: maximum time allowed to complete the symbolic execution.
- `memory`: maximum RAM that can be used. This is generally a hard constraint during symbolic execution bounded by the available resources, but it could constitute an useful tunable parameter in the case in which more Angr instances are run in parallel to analyze different binaries. Nowadays, considering that symbolic execution frameworks are usually single-threaded, many server machines will perceive first a limitation in terms of RAM than in CPU cores.
- `loop threshold`: limits the number of times the same memory address could be visited before stopping the execution of such a trace and stashing the state. If no other state is available and the timeout has not been reached, these states are resumed at a later time.

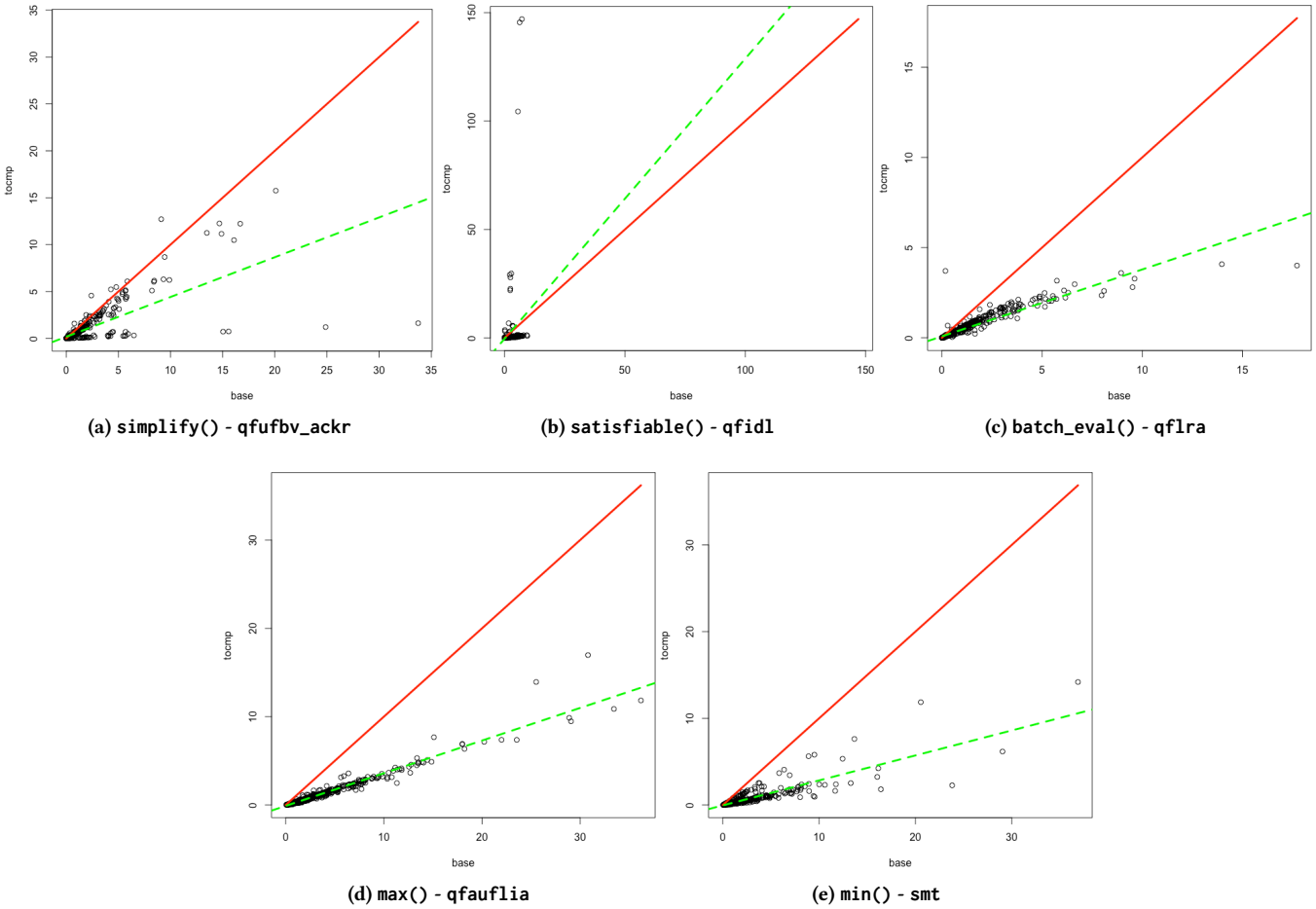
In our experiments these parameters have been set up respectively to: timeout of 1 hour, memory limit of 10 GB and no loop threshold.

There are also several heuristics that could be used to get rid of the inner limitations of the symbolic execution discussed in Section 3.3. In this experimental study, the heuristics under evaluation are:

- `step timeout`: maximum time to compute a step of the symbolic execution. This mainly corresponds to the time consumed by the SMT solver.
- `branching loop threshold`: a lightweight heuristic to identify the presence of symbolic loops and provide the possibility of stopping unrolling such loops. The difference from the above mentioned loop-threshold lies in the fact that the latter considers only if a given address is visited again but, having concrete variables, the execution trace is still unique. Instead, conditional jumps involving symbolic variables could possibly spawn new execution traces and need to call the SMT solver to check if both the branches of a conditional jumps

Table 2: Results of the selected z3 tactics over our benchmark of extracted SMT expressions. The performance of the best tactic for each Claripy call is in bold.

	simplify()		satisfiable()		batch_eval()		max()		min()	
Tactic	# solved	Exec. time								
NO_TACTIC	54460	694.22812	103881	765.75330	47585	603.7525	177439	1764.67108	112712	1087.35926
qfufbv_ackr	54460	**386.65576**	103881	1139.12232	47585	449.5783	177439	1203.88404	112712	478.72028
solve-eqs	-	-	25136	20.14145	-	-	-	-	-	-
qfnra-nlsat	642	90.17725	93194	78.75564	4783	196.3956	26419	189.45631	37930	199.52513
qfnra	642	4415.16436	93194	160.87066	4783	287.8888	26419	324.62751	37930	305.77329
qfidl	54460	3977.16036	103881	**671.98977**	47585	318.2233	177439	697.96655	112712	350.42974
qflra	54460	3968.18507	103881	2653.76949	47585	**279.9188**	177439	619.84178	112712	295.73773
qfauflia	54460	3954.05094	103881	1091.53502	47585	296.3621	177439	**605.63197**	112712	295.56502
smt	54460	3983.59492	103881	2645.59894	47585	284.2394	177439	610.80981	112712	**292.04155**
Tot expressions	54460		103881		47585		177439		112712	



(a) simplify() - qfufbv_ackr (b) satisfiable() - qfidl (c) batch_eval() - qflra

(d) max() - qfauflia (e) min() - smt

Figure 2: Execution time (in seconds) for solving the expressions in the benchmark. Each data point represents the total execution time for all the expressions from the corresponding Claripy function in a given binary. On the x-axis the NO_TACTIC, on the y-axis the selected tactic. The diagonal line represents the scenario in which both the tactic have the same execution time. The dashed line in light green represents instead a simple linear interpolation model for the correlation. Having the default tactic on the x-axis, if the green line is below the diagonal, the customized tactic outperforms the default one in terms of execution time.

are actually feasible or not. Several sophisticated loop unrolling strategies proposed in literature aim at performing an informed guess on when no useful information could be extracted from the loop such as the Loop-Extended Symbolic Execution (LESE) [43], the Read-Write set (RWset) [11] and the bit-precise symbolic loop mapping [55]. LESE [43] introduces symbolic variables counting the number of times each loop is executed and then links these variables with features of a known input grammar such as variable-length or repeating fields. When new execution paths are generated in presence of execution branches, RWset [11] tracks the memory locations read and written by the checked code to determine if the reminder of a trace can explore new behaviors and prunes it otherwise. Instead, the trace-based approach proposed in the bit-precise symbolic loop mapping [55] aims at identify the semantics of possible cryptographic algorithms used in obfuscated binary code. Differently from these techniques, our approach is much less computational demanding even though it is a pure heuristic counting only the number of visits to the same address from which a new execution trace has been spawned. Furthermore, other approaches consider to solve symbolic loops through the use of concolic execution [21] i.e., arbitrarily concertizing some of the variables.

- `max active paths`: a limit on the number of paths symbolically executed simultaneously. By default, at each execution step Angr considers all paths that are not finished yet, thus exploring the binary with a breadth-first search (BFS) strategy. If the limit is set, only a few paths are executed while all the rest are waiting in a queue. Thus, the binary is explored more in depth and potentially the memory consumption is reduced.

5.2 Methodology

To assess the classifier's ability in generalizing the predicted class (i.e., either cleanware, or malware including which malware family) for binary samples we used cross-validation. This technique allows to reduce dependency from the dataset used during the learning phase and thus to avoid overfitting. In particular, we considered *k-fold cross validation*. The approach foresees the random partition of the original dataset in k datasets of equal size, where $k-1$ of them are used for learning and the remaining one for validation. The process is repeated k times, every time changing the validation subset (so that all the samples are used for both learning and validation), and results are averaged. In our experiments k has been set to 5.

Our experiments aim at: (i) pinpointing the graph metrics corresponding to better SCDGs for malware classification; (ii) explaining how heuristics affect the quality of the traces extracted by means of symbolic execution and the graph building process. In this section, our *Design Of Experiments (DOE)* is described. As response variables we considered the *micro-average F-score* for the malware families, and the F_1 *score* for the binary classifier (see Section 2.2).

For the input variables we used a *full factorial design* in which factors correspond to the graph building heuristics (see Section 2.1), the symbolic execution heuristics (Section 5.1) and the customized set of z3 tactics identified in Section 4.2. For each factor we have considered two levels according to our past experience. Levels for each factor are summarized in Table 3. *Experimental units* were thus constituted by all the possible combinations of the two levels of each factor. Even if such a design of experiments allows one to pinpoint how the response variable is influenced by each factor and by the interactions between the latter, it is worth to take into account that the number of experimental units grows as two to the power of the number of factors (128 units in our case).

Table 3: Levels for each factor of the graph building and symbolic execution heuristics

	Factor	Levels
graph	merge-calls	True, False
	disjoint union / traces-merge	disjoint, merge
	min-trace-size	0, 10
execution	step timeout	8, $+\infty$
	branching loop threshold	4, $+\infty$
	max-active-paths	8, $+\infty$
	z3 tactics	default, optimized

Results of our comparative experiments are analyzed considering the correlations between each possible pair of variables (i.e., graph metrics, heuristics and response variables) and according to the *Analysis of Variance (ANOVA)*. In particular, the latter approach allowed one to discover the way in which the variance exhibited by a dependent variables (F-score by malware family in our case) could be due to variations of independent variables.

6 RESULTS, ANALYSIS AND DISCUSSION

In this section, we first consider the correlation between heuristics, SCDGs and quality of the malware classifier (Section 6.1) and then evaluate the interactions between heuristics and their impact to the classifier (Section 6.2).

During our experimental campaign, the best classifier scored an F-score by malware family of 0.955 and F-score for the binary classifier of 0.970 (with a similarity threshold for gSpan of 0.7). At the best of our knowledge, only few recent works have tried to classify PE malware (Windows) by their family and none of them achieved such a high quality of the classifier and none was using the symbolic execution. By extracting static features (i.e., textual and binary patterns) with YARA [2], Sun et al. [48] achieved an F_1 score of 0.936 with a random forest classifier (RF) over a dataset of 2798 malware belonging to 12 families. Still using a RF but extracting behavioral traces through a concrete execution within the Cuckoo sandbox [3], Hansen et al. [29] achieved an F-score of 0.864 on a dataset of 31295 malware samples from 5 families. Thanks to a RF fed with a mix of static and dynamic features obtained by executing and monitoring the execution of 2939 binaries from 17 families with the HookMe tool [51] (similarly to Cuckoo, a virtual environment allowing one to hook system services and extracting traces), Islam et al. [30] achieved an accuracy of 0.97 (misclassification are not

[2] https://github.com/seqan/seqan/tree/master/apps/yara
[3] http://www.cuckoosandbox.org/

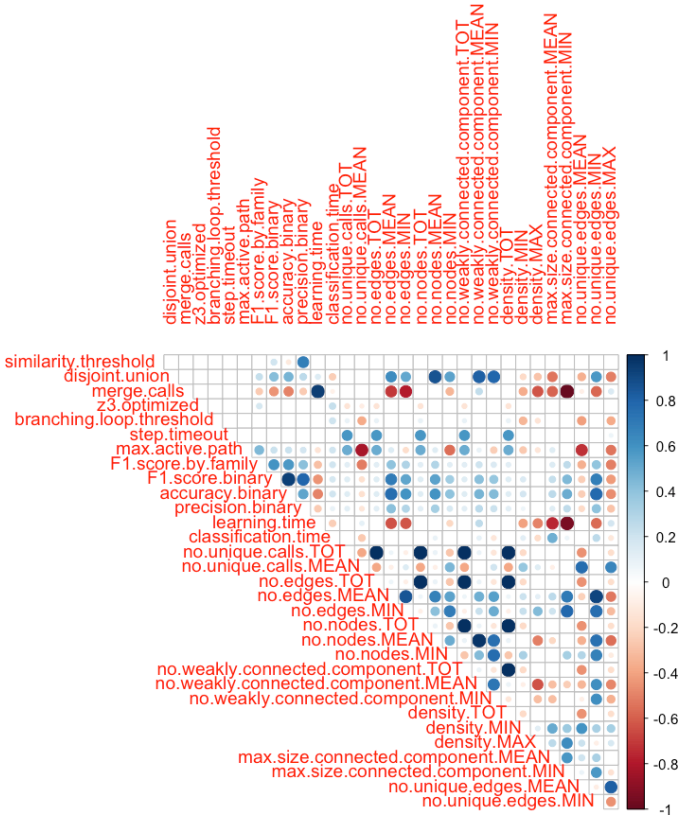


Figure 3: Correlation matrix (*p-value* < 0.01) for the graph building and symbolic execution heuristics, graph metrics and performance of the classifier (some correlations have been removed from the plot for the sake of clarity).

reported making impossible to compute the F-score, that it is lower by construction).

6.1 Correlation between Heuristics, SCDGs and Classification

The correlation between each possible pair of graph building and symbolic execution heuristics, SCDGs metrics and the quality metrics of the classifier (F-scores, accuracy and precision) are represented in Figure 3. For the graph metrics described in Section 2.1 we computed some summary statistics (total, mean, standard deviation, min, quartiles Q_1, Q_2, Q_3 and max). Evaluating the correlations we considered only the ones having significance level of at least 0.01. Size and color intensity of the circles represent how strong the correlation is (if any) between the pair of parameters indicated on the border of the matrix.

The F-score by malware family is negatively correlated to the presence of many unique edges and nodes. Moreover, it is negatively correlated to long learning times. This last observation is due to the higher time required by gSpan to analyze larger graphs. The negative correlation with the unique edges and nodes counterpoints the positive correlation with nodes, edges and connected components. This could be explained by the need of large connected sub-graphs for supporting the gSpan mining disregarding their identification as "unique" (see Section 2.1). Finally, the F-score by malware family is strongly correlated with the performance of the binary classifier with the latter having even stronger correlations with the above mentioned graph metrics.

The merge calls graph-building heuristic (described in Section 2.1) causes a significant reduction in the number of edges and size of the connected components. Even if this simplification significantly reduces the learning time, this is paid with a reduced quality of the classifier.

The use of disjoint union as trace combination heuristic causes the presence of an higher number of sub-graphs and thus requires a higher learning time. On the other hand, the presence of an higher number of connected components, as observed before, increases the F-score.

The last of the considered graph-building heuristics, the minimum trace size, does not present any significant correlation with any of the considered performance or graph metrics (and has been therefore omitted in the correlation matrix in Figure 3).

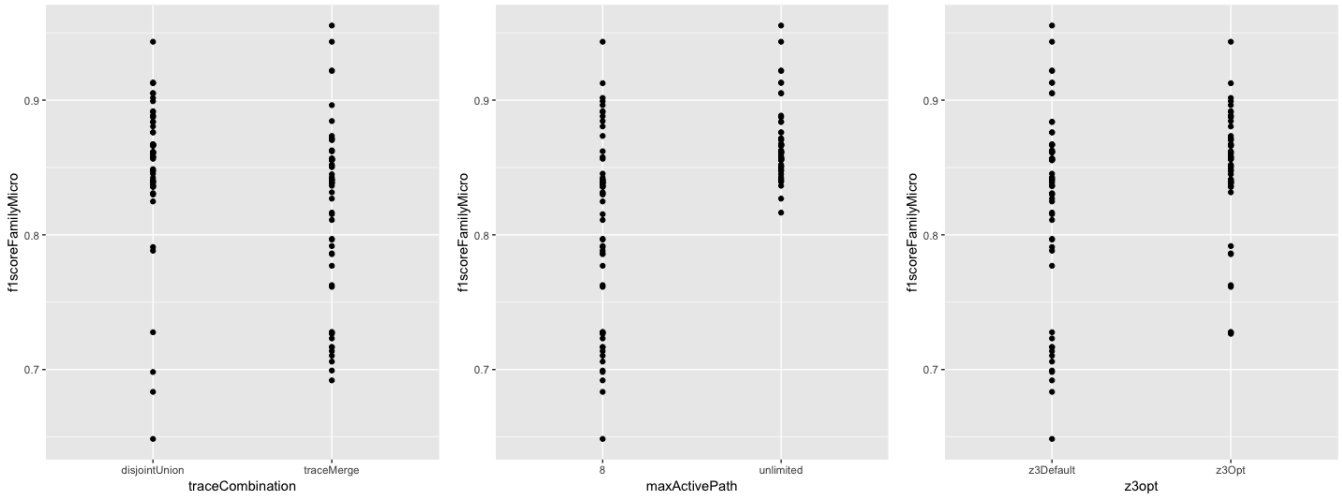
For what concerns the symbolic execution heuristics: (i) our customized set of z3 tactics has a positive correlation with the F-score (as later exploited in Section 6.2), (ii) the branching loop threshold allows to collect more unique calls and edges but this is not reflected on the F-score (that mainly depends on the connected components as already observed), (iii) the max active paths set to infinity is positively correlated with the F-score thanks to its ability to explore the CFG in a BFS order.

We can thus conclude that, considering the graph metrics and their impact on the F-score, the litmus test for the quality of an SCDG-based classifier is represented by the presence of connected components. This could be explained considering how the gSpan mining algorithm works and the adopted similarity metric based on the number of common edges between the extracted signatures and the SCDG of the sample to classify (see Section 2.2).

6.2 Impact of Graph and Execution Heuristics

In the ANOVA, we first consider the main effect of each heuristic on the quality of the classifier. For the sake of compactness in Figure 4 we show only three out of the seven studied factors. For the graph building heuristics, performing the disjoint union of the traces is generally preferable (see Figure 4a), as well as to not merge the calls (not shown in the figure). Instead, using a limit to the minimum trace size to build a graph does not have a direct impact on the classifier. This happens because when Angr is able to analyze the malware (see the following Section 6.3), generally it is able to extract a substantial number of calls. The use of our customized version of z3 allows the consolidation of the classifier to an F-score above 0.825, whereas the default version has performance highly spread even reaching a very low F-score of 0.65 (see Figure 4c). Limiting the max active paths brings to very inconstant performance from an F-score of 0.65 to 0.94. Finally, branching loop threshold and step timeout show a similar behavior where, if disabled, the performance are more constant.

We consider a linear model to study the interaction between each pair of factors. For the sake of brevity, here we only comment the significant observations of the corresponding 21 plots. Trace



(a) Trace combination **(b) `max active path`** **(c) `z3 config`**

Figure 4: Main effects of the factors over the F-score. Each plot represents the levels for a given factor on the x-axis, and the F-score on the y-axis.

merge combined with merge calls is too aggressive in simplifying the graphs bringing to catastrophic effect on the F-score when both options are used together. This is due to the fact that both reduce the number of edges and connected components, with substantial degradation on the most important graph metrics identified in Section 6.1. Even if, as already observed, the merge calls heuristic has generally negative effects on the F-score, the optimized version of z3 is able to mitigate this. The reason lies in the ability of the optimized z3 to actually determine the satisfiability of SMT constraints instead of returning unknown due to a solver timeout and thus extracting edges to connect the nodes. The effect of merge calls and branching loop threshold appears to be clearly dependent. Exactly the opposite by what is shown by the interaction plot of merge calls and step timeout. A very interesting dependency is shown between the z3 solver and the limit on the number of max active paths. Limiting the active paths is something usually done to partially reduce the memory consumption (so some performance degradation are tolerable), and also to make the symbolic execution a little more Depth-First Search (DFS) instead of BFS. Here, we observe a significant performance decrease by enabling the max active paths limit, but our optimized version of z3 is able to almost completely remove such a negative impact on the F-score (the degradation is still present but it is of about 0.01 instead of about 0.09). Other plots provide some hints of dependencies such as: trace combination with min trace size and with branching loop threshold, merge calls and min trace size, and z3 with branching loop and step timeout. This preliminary observations are examined in depth in the rest of this section.

We finally considered the interactions between each possible combination of factors. The analysis show several interaction effects asserted with a significance lower than 0.001. The Pareto plot (not shown here due to space limitations) helped us to select the ones having more influence on the F-score. The combination of trace merge, merge calls, optimized z3, and unlimited max active paths has shown the most influential positive effect (above 0.3). This is followed by the negative effect of the combination of trace merge, merge calls, optimized z3, no branching loop, and unlimited max active paths. Positive impact with a magnitude of about 0.2 on the F-score are attributed to the sets of: (i) trace merge, merge calls, disabled branching loop and unlimited max active paths, (ii) merge calls and unlimited max active paths. The merge calls heuristic takes part also to some of the subsequent sets having negative effects. Exactly the opposite of what happens for the optimized z3 (which has generally a positive impact). The main factors having the highest magnitude are indeed the previously mentioned two.

We can thus conclude that:

- merge calls is very risky, even if it is part of some of the best configuration. Similarly for the trace combination heuristic: disjoint union has more constant performance.
- max active paths, as expected, should be set to unlimited if there are no other reasons to do otherwise.
- the optimized z3 helps, even if it is not the performance leader, it supports many of the other configurations that may be need to enable due to resource constraints (e.g., the max number of active paths).
- branching loop: shows very seesawing effects according to the configuration in which it is used. This factor requires further evaluation with a higher number of levels to better understand its performance.
- step timeout and min trace size do not have a strong effect on the F-score.

The results of the factorial experiments show that in our context tuning the symbolic execution is a very complex problem and that the *sparsity of effect principle* (stating that the system is dominated by the effect of the main factors and low-order-factor interactions) does not hold.

6.3 Threats to validity

In this section we discuss about the validity [41] of our study according to: construct, internal, external validity and reliability.

Construct validity concerns to the employed technique and if it is able to actually investigate the properties of interest. In our case we have performed controlled experiments that allowed us to tune the environment and measure the metrics of interest. At this regard, no issue should arise from our experiential study. It is worth to remark that the presented exploratory study concerns more on exposing which factors are most likely to influence the outcome than on setting up a quantitative model.

Internal validity considers casual relations under analysis and if there could be other factors not considered in the experiments. In our experimental study we performed a full factorial experiment design with the aim of get rid of this threat to validity even if we had some a-priori hypothesis on the most effective heuristics (e.g., the branching loop threshold and the z3 optimization) from previous experiments.

External validity deals to the extent our finding can be generalized. This aspects is the most critical one for our experimental study. About the identified most important heuristics to extract good SCDGs, we considered a single symbolic execution framework (i.e., Angr). Even if several other tools provide similar heuristics it is not possible to take for granted that the same are available in every framework e.g., S^2E does not provide path selection heuristics, contrarily to Triton and Angr, but it is possible to write plugins to add such a functionality. For what concerns the adopted malware dataset, several issues may potentially affect our results. Our dataset is composed by 225 executable binaries in PE format and consisting of 8 malware families. Limits on number of samples and families of the considered malware do not allow to draw very general conclusions. In particular, the family classification is very inconsistent among the antivirus products. The same holds for what concerns the malware type. In many cases a multi-labeling seems the most appropriate approach. The presence of packed binaries is another critical aspect of our analysis approach. The symbolic execution framework (i.e., Angr) could spend time analyzing the packer instead of its binary payload therefore extracting very similar SCDGs. In our case, a post-analysis examination with VirusTotal allowed us to assess that all and only the samples in the *addrop* family were packed with NSIS. Therefore, even if in this case we most probably limited our exploration to the packer, the rest of the analysis has not been affected. In a future work, we will consider the implementation of an unpacker module to be used before starting the actual symbolic execution, similarly to the solution adopted by PyREBox [49]. Moreover, more general limits affecting symbolic execution frameworks could limit the applicability of the considered malware detection approach. In particular, in the case of Angr, it can evaluate only binaries, therefore malware in the form of tampered document files (e.g., pdf) or scripts (e.g., JavaScript or PowerShell) could not be analyzed. Furthermore, Angr provides models for a few system calls, simulated using custom functions in Python, lack of models for external libraries or system calls prevent it to successfully analyze the binary. Same limits are also present in case of particular input parameters or network resources are required to start the execution. Finally, if the malicious behavior is hidden behind system signals or exceptions [57], we are not able to keep track of it with the current version of Angr.

In this context, *reliability* concerns the reproducibility of experiments and analysis obtaining the same results. Assuming the use of a similar set of malware families (and a family classification consistent with ours, a very critical point indeed), having adopted the ANOVA test and k-fold validation obtaining evidence with a low significance level (0.001), our findings should be reasonably congruent with the ones hypothetically drawn by other researchers.

7 CONCLUSION

Static symbolic execution is a very powerful tool in the struggle against malware faced by security analysts. During symbolic execution, complex constrained expressions are built in presence of conditional jumps. Unfortunately, obfuscation techniques used by malware exacerbate the already demanding resource consumption of this analysis approach and thus hamper its use in practice.

In this experimental work, we (i) identified as the main SCDG quality metric the connected components, (ii) explored the impact of several graph-building and symbolic execution heuristics through a full factorial experiment design.

In particular, the SMT solver z3 has been optimized building a benchmark of constrained expression in SMT-LIB v2 format extracted from the symbolic execution of a dataset constituted by about 900 malware and cleanware. The evaluation proved that this is the most influential positive factor also showing an ability in reducing the impact of heuristics that may need to be enabled due to resource constraints (e.g., the max number of active paths).

The impact of some of the heuristics broadly implemented in symbolic execution frameworks has been thoroughly evaluated with a factorial experiment approach. Results suggest that the most important factors are the disjoint union (as trace combination heuristic), and the z3 optimization whereas other heuristics (such as min trace size and step timeout) have less impact on the quality of the constructed SCDGs.

Notably, our experimental campaign allowed us to achieve an F-score of 0.955 for the malware family classification (and an F-score of 0.970 for the binary classifier), improving previous literature results.

Our analysis has been limited to the Angr framework albeit the features considered during our optimization (i.e., SMT solver and heuristics) are available in many binary symbolic execution tools and thus our results should be reasonably applicable to them as well. To further improve the performance of the SMT solver we are investigating the use of Genetic Algorithm (GA) approaches, like the evolutionary algorithm proposed in [28] to combine tactics in z3 for linear problems.

In our work, the classifier adopts as similarity metric the number of edges in common between the subgraphs found in the graphs of the binary under evaluation and the semantic signatures of the malware. More sophisticated similarity measures such as the euclidean distance after having represented nodes in a 2-D space according to their inner and outer degrees [37] could be adopted. As a future work, we are evaluating a refinement of such a metric considering ideas from [32] which addresses a similar problem (i.e., how to calculate the similarity of two graphs possibly having different

number of edges and nodes) and [15] (considering labeled nodes and edges).

Analysis with a larger dataset in terms of malware families and samples are also required to better assess the performance of the classifier. Another important aspect concerns the ability of such an approach in classifying the malware also according to their type. Given a generalized inconsistence among the antivirus products in classifying malware family and type, a multi-labeling approach is deemed the most appropriate one.

ACKNOWLEDGMENTS

The authors would like to thank *VirusTotal.com* for providing malware samples and access to their API.

REFERENCES

[1] 2012. *NIST/SEMATECH e-Handbook of Statistical Methods*. NIST. http://www.itl.nist.gov/div898/handbook/

[2] Shabnam Aboughadareh, Christoph Csallner, and Mehdi Azarmi. 2014. Mixed-Mode Malware and Its Analysis. In *PPREW-4*.

[3] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *USENIX Security Symp. 2017*.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Assisting Malware Analysis with Symbolic Execution: A Case Study. In *CSCML 2017*, Shlomi Dolev and Sachin Lodha (Eds.).

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2016. A Survey of Symbolic Execution Techniques. *CoRR* abs/1610.00502 (2016). arXiv:1610.00502 http://arxiv.org/abs/1610.00502

[6] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code Obfuscation Against Symbolic Execution Attacks. In *ACSAC '16*.

[7] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0.

[8] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. 2010. Improving the Efficiency of Dynamic Malware Analysis. In *SAC '10*.

[9] J. Bellizzi and M. Vella. 2015. WeXpose: Towards on-line dynamic analysis of web attack payloads using just-in-time binary modification. In *ICETE 2015*.

[10] Najah Ben Said, Fabrizio Biondi, Vesselin Bontchev, Olivier Decourbe, Thomas Given-Wilson, Axel Legay, and Jean Quilbeuf. 2018. *Detection of Mirai by Syntactic and Semantic Analysis*. Technical Report. Inria. https://hal.inria.fr/hal-01629040

[11] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *TACAS 2008*.

[12] Alexei Bulazel and Bülent Yener. 2017. A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web. In *ROOTS 2017*.

[13] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. 2012. A Quantitative Study of Accuracy in System Call-based Malware Detection. In *ISSTA 2012*.

[14] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *SP '12*.

[15] Pierre-Antoine Champin and Christine Solnon. 2003. Measuring the Similarity of Labeled Graphs. In *ICCBR'03*.

[16] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. *SIGPLAN Not.* 47, 4 (March 2011), 265–278. https://doi.org/10.1145/2248487.1950396

[17] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. 2007. Mining Specifications of Malicious Behavior. In *ESEC-FSE '07*.

[18] Christian Collberg. 2012. The Tigress C Diversifier/Obfuscator. http://tigress.cs.arizona.edu. [Online accessed 17-May-2018].

[19] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. 2012. Distributed Application Tamper Detection via Continuous Software Updates. In *ACSAC '12*.

[20] K. H. T. Dam and T. Touili. 2016. Automatic extraction of malicious behaviors. In *MALWARE 2016*.

[21] Robin David. 2017. *Formal Approaches for Automatic Deobfuscation and Reverse-engineering of Protected Codes*. Ph.D. Dissertation. Université de Lorraine.

[22] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS 2008*.

[23] Leonardo de Moura and Grant Olney Passmore. 2013. *The Strategy Challenge in SMT Solving*. Springer Berlin Heidelberg, Berlin, Heidelberg, 15–44. https://doi.org/10.1007/978-3-642-36675-8_2

[24] Marko Dimjašević, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamaric. 2016. Evaluation of Android Malware Detection Based on System Calls. In *IWSPA '16*.

[25] Chris Eagle. 2008. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, San Francisco, CA, USA.

[26] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2008. A Survey on Automated Dynamic Malware-analysis Techniques and Tools. *ACM Comput. Surv.* 44, 2, Article 6 (March 2008), 6:1–6:42 pages. https://doi.org/10.1145/2089125.2089126

[27] Matt Fredrikson, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. 2010. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. In *SP '10*.

[28] Nicolás Gálvez Ramírez, Youssef Hamadi, Eric Monfroy, and Frédéric Saubion. 2016. Towards Automated Strategies in Satisfiability Modulo Theory. In *EuroGP 2018*.

[29] S. S. Hansen, T. M. T. Larsen, M. Stevanovic, and J. M. Pedersen. 2016. An approach for detection and family classification of malware based on behavioral analysis. In *ICNC 2016*.

[30] Rafiqul Islam, Ronghua Tian, Lynn M. Batten, and Steve Versteeg. 2013. Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications* 36, 2 (2013), 646 – 656. https://doi.org/10.1016/j.jnca.2012.10.004

[31] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. 2017. VMAttack: Deobfuscating Virtualization-Based Packed Binaries. In *ARES '17*.

[32] Danai Koutra, Ankur Parikh, Aaditya Ramdas, and Jing Xiang. 2011. *Algorithms for Graph Similarity and Subgraph Matching*. Technical Report. Carnegie Mellon University. https://people.eecs.berkeley.edu/~aramdas/reports/DBreport.pdf

[33] Nikolay Kuzurin, Alexander Shokurov, Nikolay Varnovsky, and Vladimir Zakharov. 2007. On the Concept of Software Obfuscation in Computer Security. In *ISC 2007*.

[34] Hugo Daniel Macedo and Tayssir Touili. 2013. Mining Malware Specifications through Static Reachability Analysis. In *ESORICS 2013*.

[35] Hugo Daniel Macedo and Tayssir Touili. 2013. Mining Malware Specifications through Static Reachability Analysis. In *ESORICS 2013*.

[36] Smita Naval, Vijay Laxmi, M. S. Gaur, and P. Vinod. 2012. SPADE: Signature Based PAcker DEtection. In *SecurIT '12*.

[37] Stavros D. Nikolopoulos and Iosif Polenakis. 2017. A graph-based model for malware detection and classification using system-call groups. *Journal of Computer Virology and Hacking Techniques* 13, 1 (01 Feb 2017), 29–46. https://doi.org/10.1007/s11416-016-0267-1

[38] Sirinda Palahan, Domagoj Babić, Swarat Chaudhuri, and Daniel Kifer. 2013. Extraction of Statistically Significant Malware Behaviors. In *ACSAC '13*.

[39] Chinmaya Kumar Patanaik, Ferdous A. Barbhuiya, and Sukumar Nandi. 2012. Obfuscated Malware Detection Using API Call Dependency. In *SecurIT '12*.

[40] Radare2 Team. 2017. *Radare2 Book*. GitHub.

[41] Per Runeson and Martin Höst. 2008. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (19 Dec 2008), 131. https://doi.org/10.1007/s10664-008-9102-8

[42] Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC*.

[43] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended Symbolic Execution on Binary Programs. In *ISSTA '09*.

[44] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. *SIGSOFT Softw. Eng. Notes* 30, 5 (Sept. 2005), 263–272. https://doi.org/10.1145/1095430.1081750

[45] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *SP 2018*.

[46] VMProtect Software. 2018. VMProtect - new-generation software protection. Retrieved May 17, 2018 from http://vmpsoft.com/products/vmprotect/

[47] Fu Song and Tayssir Touili. 2012. Pushdown Model Checking for Malware Detection. In *TACAS 2012*.

[48] B. Sun, Q. Li, Y. Guo, Q. Wen, X. Lin, and W. Liu. 2017. Malware family classification method based on static feature extraction. In *ICCC 2017*.

[49] Cisco Talos. 2017. PyREBox: Python scriptable Reverse Engineering sandbox. Retrieved Aug 12, 2018 from https://github.com/Cisco-Talos/pyrebox

[50] Oreans Technologies. 2017. Themida: Advanced windows software protection system. Retrieved May 17, 2018 from http://www.oreans.com/themida.php

[51] R. Tian, R. Islam, L. Batten, and S. Versteeg. 2010. Differentiating malware from cleanware using behavioural analysis. In *MALWARE 2010*.

[52] C. Willems, T. Holz, and F. Freiling. 2007. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security Privacy* 5, 2 (March 2007), 32–39. https://doi.org/10.1109/MSP.2007.45

[53] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Moura. 2013. Efficiently Solving Quantified Bit-vector Formulas. *Form. Methods Syst. Des.* 42, 1 (Feb. 2013), 3–23. https://doi.org/10.1007/s10703-012-0156-2

[54] Zhenyu Wu, Steven Gianvecchio, Mengjun Xie, and Haining Wang. 2010. Mimimorphism: A New Approach to Binary Code Obfuscation. In *CCS '10*.

[55] D. Xu, J. Ming, and D. Wu. 2017. Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping. In *SP 2017*.
[56] Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *CCS '15*.
[57] Babak Yadegari, Jon Stephens, and Saumya Debray. 2017. Analysis of Exception-Based Control Transfers. In *CODASPY '17*.
[58] W. Yan, Z. Zhang, and N. Ansari. 2008. Revealing Packed Malware. *SP '08* (2008).
[59] Xifeng Yan and Jiawei Han. 2002. gSpan: graph-based substructure pattern mining. In *ICDM 2002*.
[60] M. Yusuf, A. El-Mahdy, and E. Rohou. 2013. On-stack replacement to improve JIT-based obfuscation a preliminary study. In *JEC-ECC 2013*.