Optimizing Symbolic Execution for Malware Behavior Classification

Stefano Sebastio[a,*], Eduard Baranov[b], Fabrizio Biondi[c], Olivier Decourbe[a], Thomas Given-Wilson[b], Axel Legay[b], Cassius Puodzius[a], Jean Quilbeuf[d]

[a] *Inria, Rennes, France*
[b] *Universite Catholique de Louvain, Louvain-la-Neuve, Belgium*
[c] *Avast, Prague, Czechia*
[d] *Cisco, Paris, France*

Abstract

Increasingly software correctness, reliability, and security is being analyzed using tools that combine various formal and heuristic approaches. Often such analysis becomes expensive in terms of time and at the cost of high quality results. In this experience report we explore the tuning and optimization of the tools underlying binary malware detection and classification. We identify heuristics and SMT solver tactics for the effective symbolic execution of binary files. We combine these with effective heuristics for the construction of behavioral signatures of programs that can be used for a supervised learning multi-class malware classifier. Further, a set of experiments following the full-factorial design allowed us to identify the correlations between heuristics and the overall performance of the classifier.

Keywords: Malware classification, Empirical studies, SMT solving, Behavior graphs, Symbolic execution

1. Introduction

Increasingly software systems have their correctness, reliability, and security analyzed by tools that combine various formal and heuristic approaches [1, 2, 3, 4, 5]. The tools that make use of these approaches include: compilers [6, 7, 8], code analyzers [9, 10, 11, 12, 13], disassemblers [14, 15, 16], and binary analysis engines [17, 18, 19] to name some examples.

These tools combines their use of formal methods that attempt to provide complete and correct coverage, with heuristics that try to optimize and approximate when formal methods alone are too expensive or intractable. In practice these tools often come with hundreds or thousands of configuration and tuning

*Corresponding author
Email address: stefano.sebastio@inria.fr (Stefano Sebastio)

options that can be used to adjust their behavior to try and improve their overall effectiveness. However, since many rely on other tools to solve sub-problems or are applied in many scenarios, typically the configuration is left to default or an imprecise guess.

This work considers the problem of program behavior analysis and the specific domain of malware analysis as a case study that raises many interesting challenges for tools. The main challenge here is to work from the program *binary* to accurately determine program *behavior*, in particular behavior that may be obfuscated to avoid such analysis. Indeed, implementing anti-analysis obfuscation techniques is common [20, 21, 22] including: virtualization [23, 24, 25, 26, 27], Just-In-Time compilation [27, 28, 29] and packing [30, 31, 32]. Further, polymorphism [33, 34, 35, 36] and metamorphism [37, 38, 39] target signature-based malware detection. Thus, malware exemplifies both the key aspects of program analysis, and is in a domain where the challenges are among the hardest.

The analysis of (malware) program behavior is usually distinguished as *dynamic* or advanced *static* analysis. In dynamic analysis [40] (malware) programs are executed in an isolated environment to understand how they behave [41]. The main limitation is that the analysis strongly depends on the context emulated by the isolated environment (*sandbox*) making it hard to guess how to trigger the malicious behavior of the malware, and various techniques can easily hinder dynamic analysis [42]. Static techniques [43] instead perform the analysis of possible malware execution paths by exploring the code without executing it. To analyze malware its binary must be disassembled and then static code analysis performed *concretely* or *symbolically*.

In concrete analysis, the execution trace over the disassembled code is guided by provided contextual information and therefore has similar drawbacks to dynamic analysis: the exploration is limited by the chosen context. In symbolic analysis, *symbolic execution* traverses the code considering symbolic input variables in place of concrete values. Thus, all the possible execution paths are taken into account and the Control Flow Graph (CFG) can be (potentially) fully explored. When different branches of execution are encountered the program state is replicated for each new branch and, in order to control the execution, a set of constraints on symbolic variables is tracked for each state. These constraints create *Satisfiability Modulo Theories* (SMT) decision problems solved with the aid of a *theorem prover* to understand if a given execution path is feasible. In addition to these formal approaches, various heuristics are used to guide and optimize symbolic execution.

When considering the behavior of a program, software interacts with their host system through *System Calls (SCs)* [44]. The *System Call Dependency Graph (SCDG)* built from the information flow propagated through these SCs (originally proposed in [45, 46]) has been proved to be a good data structure to represent a *behavioral signature* of a software sample [47, 48, 49], and thus to classify malware from cleanware [50, 51, 52, 53].

Our analysis toolchain for malware classifier construction with SCDGs is represented in Figure 1 and operates as follows. A collection on binaries in-

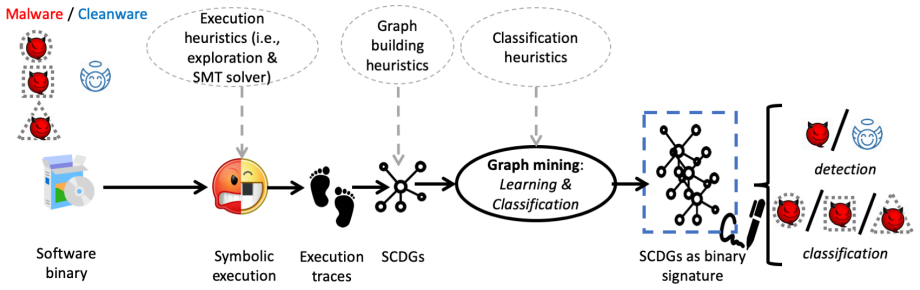


Figure 1: Analysis toolchain for malware behavioral classifier construction using symbolic execution. The binary under analysis (whether malicious or clean) is executed symbolically using Angr. Execution heuristics are used to optimize the SMT solver and execution path exploration. Execution traces are generated by Angr and are then converted into SCDGs using various graph building heuristics. These SCDGs represent how the sample interacts with the host system. Then a classifier is constructed using the SCDGs as learning inputs to create signature graphs for each malware family. By means of graph matching, the classifier is able to perform malware *detection* (i.e., discern if the binary under analysis is malicious or not) and *classification* (i.e., identify the family to which the malware belongs to). The dashed contour lines around the malware icons identify different malware families.

cluding several malware families as well as cleanware is used as the input to the toolchain. These binaries are given to Angr to produce symbolic execution traces. Due to limitations on the available resources, several execution heuristics are used to optimize the symbolic execution. From a single binary, multiple execution traces can be generated by following different execution branches. Execution traces are further transformed into SCDGs extracting an abstract behavioral representation. Several graph building heuristics are used to improve the resulting SCDGs and to optimize the transformation. These SCDGs represent how the sample interacts with the host system. Finally, the SCDGs are used in a supervised machine learning *multi-class malware classifier*[1]ote that in this work we are interested in the *multi-class* (also referred as *multinomial*) and in the *binary* classification not in *multi-label* classification. All kinds of classification are interesting in the malware domain. Multi-class aims at classifying instances in presence of more than two classes (i.e., family). In contrast to the multi-class classification, a binary classifier can only detect if a sample is a malware or a cleanware.based on graph similarity. Instead, the multi-label aims to attach several labels to each instance according to the different groups the sample belongs to.

In this work, the goal of the binary analysis is not only the *malware detection*, but also the *malware classification*. Whereas malware detection is the problem of determining whether a given sample is malicious, malware classification is the problem of assigning the correct malware type and family/strain to a given malicious sample. Knowing the exact type and family of a malware sample al-

[1]N

lows security companies to associate the sample with specific malicious behavior and malware campaigns. This is fundamental to protect users and maintain a functional security infrastructure and ecosystem.

For instance, when an antivirus detects an infection on a system, knowing the exact type and family of the infective malware allows properly informing the user about the threat faced and on the appropriate steps to disinfect the system (and often even disinfecting the system automatically). The classification is fundamental to threat intelligence, allowing security companies and law enforcement agencies to track specific malicious actors and distribution campaigns, and to efficiently exchange strain-specific information on how to detect new malware. This usually takes the form of strain-specific YARA rules [54] or standardized languages like STIX and TAXII [55]. A direct example is the *No More Ransom* project [56], an international effort by security companies and law enforcement agencies to help ransomware victims to decrypt their systems without having to pay the ransom demanded by criminals (with non negligible ethical implications). The utility uses malware classification to determine the exact family and version of ransomware infecting the system, and to point towards the appropriate decryptor (whenever possible).

Finally, including information about types and strains of malware samples improves automated malware detection as well, as shown recently by Rudd et al. [57]. Intuitively, providing a machine learning system with information about malware types and strains helps it embed sample features into a more meaningful topology, improving its effectiveness in discriminating malware from cleanware.

This work uses binary analysis toolchain to explore various choices in formal approaches and heuristics used in this analysis. *The goal is to better understand how to use these tools in the domain of representing malware behavior and classifying based upon this representation. This provides insight into how to build more effective malware classification engines, but also more generally on how to tune the key components of such toolchains for program analysis.* Our approach to optimize the binary behavior analysis toolchain for malware detection and classification is sketched in the workflow in Fig. 2. We have build a binary dataset consisting of malware and cleanware that have been processed according to a k-fold cross-validation (a re-sampling statistical method used to evaluate machine learning models on a limited dataset). For the purpose of optimizing the SMT solver, a benchmark of SMT expressions extracted by intercepting the Angr's call to the SMT solver (ref. no. 1 in Fig. 2) has been built. SMT tactics available in the theorem solver z3 have been analyzed and combined with the purpose of finding the sweet-spot between accuracy and performance (as detailed in Section 2). The SMT optimizations have thus been integrated in Angr. Then, implementation and optimization of binary exploration (Section 4.2) (ref. no. 2 in Fig. 2) and SCDGs building (Section 3.1) (ref. no. 3 in Fig. 2) heuristics have been considered. Finally, such a holistic optimization of the symbolic execution, graph-building, and classification heuristics in the context of malware detection and classification has been evaluated thanks to

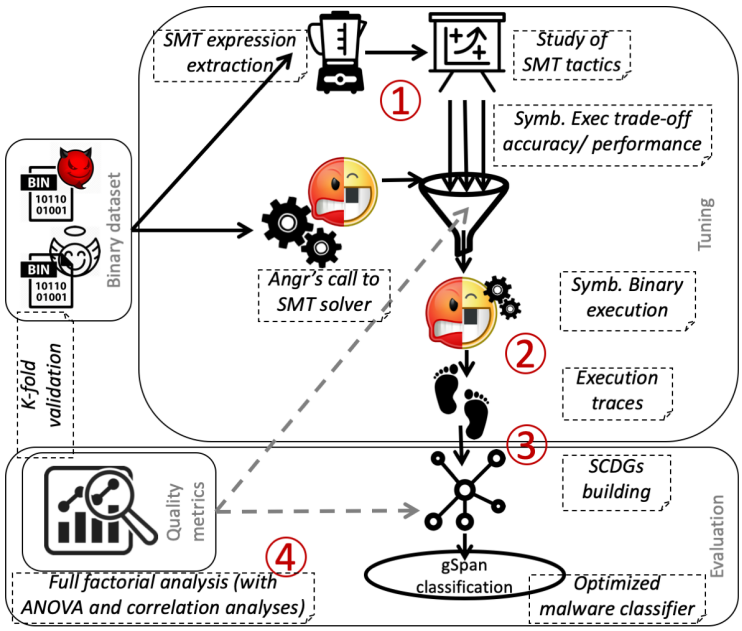


Figure 2: Workflow to perform a holistic optimization of all the components involved in malware classification using symbolic execution. From top-left: binary dataset made of malware and cleanware; extraction of SMT expressions and tuning with z3 using various tactics; optimization of the symbolic execution framework Angr; and classification of SCDG-based signatures using gSpan. The quality metrics are used to provide feedback from the evaluation back to the tuning.

a full factorial design of experiments (with ANOVA and correlation analyses) (Section 4) (ref. no. 4 in Fig. 2). Such a first-of-a-kind attempt to assess the impact of the different configuration options on efficiency and efficacy of malware classification using symbolic execution allowed us to give insights (see Section 5.4) on the interactions between the various components constituting the behavioral analysis toolchain.

The contributions of the paper are below, each addressing a specific *research question* (RQ).

- How to effectively prioritize active states in loops to reduce state explosion problems in Angr? **(RQ1)**
- How to tune the SMT solver to improve performance of symbolic execution of binaries? **(RQ2)**
- How to build graphs to effectively represent behavioral signatures? **(RQ3)**
- Can SCDG-based signatures extracted by symbolic execution be effective in malware classification? **(RQ4)**
- Are there relations between binary exploration, behavioral signatures, and classification outcomes? **(RQ5)**

The structure of the paper is described below and following Fig. 2. Section 2 overviews symbolic execution in the tuning parts of Fig. 2. Section 3 describes malware classification with behavioral signatures and presents new algorithms used in the evaluation part of Fig. 2. Section 4 details the experimental methodology and datasets used to address the RQs. Section 5 presents the results, analysis and discussion of the experiments. Section 6 presents related work on symbolic execution, SMT optimization, binary exploration heuristics and malware classification. Section 7 concludes the work with some final remarks and outlines future research efforts.

2. Symbolic Execution of Binaries

This section briefly describes the exploration of binaries with symbolic execution and identifies its key limitations. This work uses Angr [58] for the binary analysis framework, although the concepts apply to other tools such as Mayhem [17], S^2E [18] and Triton [19].

Angr translates a platform-dependent binary format into an assembly-like abstract Intermediate Representation language *Vex* [58]. Platform-dependent representations of variables, constants, and registers are converted by Vex in a consistent abstract representation by means of bitvectors [58].

Starting from the instruction at the entry point address, at each step Angr processes and updates a set of *active states* representing execution branches. Each active state stores information about the current instruction and its address as well as constraints on values of variables, registers and memory. For each active state, based on the corresponding instruction, the Angr execution

engine searches for the next states. For example, in the case of an add instruction, there would be a single next state with an updated constraint on the register into which the result is written. Another example in the case of a conditional jump instruction, there could be two next states, corresponding to making and not making the jump, each with updated constraints based on the value that the jump was conditioned upon.

One important efficiency concern is to remove infeasible states from Angr's exploration. For example, at a conditional jump the condition may be infeasible (given the constraints for the active state) and so only one next state can be executed. To determine when states are infeasible, Angr calls an SMT solver and prunes infeasible execution paths. In Angr this is done by using the z3 [59] solver to answer SMT decision problems with results of: *sat* (satisfiable), *unsat* (unsatisfiable), or *unknown*. Typically an unknown result is due to z3 failing to find a sat or unsat solution in reasonable time.

The following requests to the SMT solver are used in Angr:

- `simplify()`: simplify constraints for a subsequent satisfiability check
- `satisfiable()`: checks the constraints assigned to the current state
- `batch_eval()`: evaluates a list of constraints and provides a solution
- `max()/min()`: max/min values respecting the constraints.

The main limitation of symbolic execution in practice is state-space explosion (and this causes problems with consumption of resources such as memory, time, etc.). Thus the best solution to these problems is to optimize the usage of the SMT solver to both: provide an answer faster, and provide a `sat` or `unsat` answer instead of unknown. This in turn improves the accuracy of the exploration, and removes infeasible paths simplifying the exploration. Another viable strategy is *concolic* execution (a portmanteau word of concrete and symbolic) where symbolic variables are used only when strictly needed and concrete values otherwise [60, 61].

Other limitations specific to binary analysis are as follows.

- *Stubs* (that describe the number and bit-length of the input/output parameters) or *prototypes* (that provide bare-bone implementations modeling the behavior) of external libraries are required to correctly explore binaries. Lack of these (or the use of wrong version for such libraries) could potentially lead to an incorrect state. Currently, Angr has about 100 prototypes and 3500 stubs for Windows library API such as advapi, glibc and kernel32.

- Due to the creation of new states on each conditional branch, loops are a particular concern for Angr due to potential state explosion when loops have many iterations. To address this here we consider heuristics to detect potential state explosion in Angr and reduce the number of active states. In practice, this is achieved by detecting how many times an instruction

has been visited, and if this goes over a threshold then *stashing* this active state. Stashed states are given low priority for Angr to execute, but may be executed if no other active states are available. The first *research question* ***(RQ1)*** *is: how to detect loops to effectively improve the exploration of Angr?*

Symbolic execution faces an *accuracy/performance trade-off*: more resources (time, memory, etc.) potentially lead to better exploration. Thus the optimization of core components of the symbolic execution such as the utilization of the SMT solver are pivotal to improving the results.

The z3 theorem prover used by Angr is equipped with a few hundred parameters grouped in modules. 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 constraint 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) to tackle them. Even when the class of problem for an expression is known, a tailored tactic is not guaranteed to produce a benefit in outcome since tactics are heuristic strategies and not proven optimizations.

Tactics tend to be highly tuned for known classes of problems but may perform poorly for new classes of problems [62]. New tactics can be easily defined considering available parameters, and also by combining other tactics. A total of eight combinators of tactics known as *tacticals* [62] are available in z3. Tacticals operate by applying a set of tactics: sequentially, in parallel, or alternatively when a prior tactic fails.

For example, to simplify constraints Angr uses a customized tactic built as a sequential application of five predefined tactics provided by z3. These five tactics are: `simplify`, `propagate-ineqs`, `propagate-values`, `unit-subsume-simplify`, and `aig`, and respectively correspond to: simplification, removal of the inequalities, constant propagation, simplification of unit clauses (i.e., the ones composed by a single boolean literal), and compression of Boolean formulas.

When no tactic is provided (named `NO.TACTIC` hereafter), the z3 solver will try to infer the logic to which the expression belongs to and to apply the corresponding tactic (if available).

Previous works have considered the optimization of the solver for a specific class of formulas e.g. quantified bitvector [63] or linear integer/real arithmetics [64]. Instead, in our work we do not have any a priori knowledge about the expressions generated during the symbolic execution. It is important to note that, once instantiated, the z3 solver cannot change tactics. In the beginning of symbolic execution Angr instantiates a solver that is used for all SMT requests. Thus, we are looking for a tactic (or a set of them combined by means of tacticals) that is expected to work fast on all types of Angr's requests.

The *research question* ***(RQ2)*** *here is: can the tactics of z3 be tuned to improve performance of symbolic execution?*

3. Malware Classification via SCDG

In the previous section, we considered binary exploration for the effective extraction of execution traces, in this section we delve into the construction of behavioral signatures of binaries and how to use these signatures for malware classification.

System call dependency graphs (SCDGs) have been proved as a good approach to build binary behavioral signatures [45, 46, 47, 48, 49, 51, 52, 50]. Thanks to a common subgraph mining algorithm such as gSpan [65], it is possible to evaluate the similarity between the SCDG representing the behavior of a new binary sample and the SCDGs built from the binaries 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 malware (including family) or cleanware.

3.1. SCDG Building Heuristics

Execution traces consist of sequences of low-level instructions including system calls each with their address and arguments. In describing the binary behavior, we focus only on system calls and their relationships. Source address, arguments, and position in the trace allow one to understand the relations between the calls and thus to build a directed graph. In SCDGs vertices represent the system calls, edges the information flows between calls (i.e., one of the input or output arguments of a call is the input of another call), and direction reflecting order in the trace. In the case of system calls to the same function from different addresses, different vertices are created. Since not all the system calls have information flows in common, from each execution trace a graph with several (weakly) connected components is usually built.

Symbolic execution considers all possible execution paths of a binary, therefore several execution traces can be generated corresponding to the conditional branches that have been evaluated `sat` by the constraint solver. From these execution traces, SCDGs can be built in different ways. In particular, we consider three parametric heuristics:

- *merge-calls*: whether system calls having same name and source address (i.e., the address of the instruction calling an OS function) are merged or not. Merging the calls has the potential of providing a higher level of abstraction over the binary behavior by considering a more compact model and simplifying the graph traversing in the machine learning classification phase. On the other hand, such an abstraction could cause the loss of some details discriminating malware families.

- *disjoint union/traces-merge*: each trace from Angr can be turned into a graph. The disjoint union of these graphs is then used to contain all the behavior in a single graph (as in [50]). The alternative is to merge prefixes of traces to connect more components reducing the overall graph size (similarly to system call dependency trees of [47]). The presence of this heuristic lies in the generation of a set of traces from a single symbolic

execution. Indeed, differently from a dynamic execution where a single execution path is explored (i.e., all the variables are instantiated), in the symbolic case there is the compelling need of treating multiple traces due to the parallel exploration of all the paths. The *traces-merge* tries to generate more connected graphs (by looking for common prefixes in the set of traces), whereas the *disjoint union* does not make this effort treating the traces as independent executions (more similarly to what happens in the case of multiple dynamic executions of the same binary in a sandbox).

- *min-trace-size*: a minimum number of calls have to be present in a trace to be valid, i.e., discard shorter traces. The rational behind the use of such a heuristics is that very short traces could prompt symbolic explorations of bad quality e.g., due to anti-analysis or obfuscation techniques. Discarding such traces could prevent to bias the classifier over SCDGs representing not the behavior of a malware family itself but only the obfuscated header (e.g., in the case of packed binaries).

Recalling that execution branches yield to the generation of a set of execution traces, and that from the same set different SCDGs can be generated according to the instantiation of the graph building heuristics: the SCDG building heuristics directly affect the malware classifier. The choices of these metrics yields an interesting *research question **(RQ3)**: which graphs provide the best behavioral signatures?*

Other available SCDG building heuristics not considered here (according to our past experience) are: *max-trace-size* and *total-size-limit*. We have not explored these two because both constitute heuristics that need to be enabled only in presence of resource constraints. Therefore, keeping these heuristics unbounded, as much as possible according to our experimental setting, allowed us to remove their presence as a limiting factor on the performance of the classifier.

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

Observe that the efficiency and effectiveness of the Angr symbolic execution affect the quality of the collected information contained in the traces. The SCDG-based behavioral signature building and mining process (described in this section) determine how such information is used. The above graph metrics are later used to understand the feedback that can be gained from the *evaluation* (**RQ3**). In turns this knowledge has been exploited in the *tuning* part as shown in Fig. 2

3.2. SCDG Classification and Evaluation

The learning and classification approach presented here exploits mining of common behaviors for malware families using gSpan. This section presents how to define a multi-class classifier for malware families using gSpan.

For each family of malware, the collection of SCDG behavioral signatures for that family is mined for common subgraphs using gSpan. (Cleanware are not considered a family since they do not have a common behavior.) The largest common subgraph found from mining becomes the SCDG-based behavioral signature for that malware family.

To classify a new binary by its SCDG-based behavioral signature, a comparison is done with each family's SCDG-based behavioral signature. This comparison is done by using gSpan to find the largest subgraph of the sample and family's signature, i.e., how much of the malware family's signature is contained in the sample. We then compute the *similarity score* as the percentage of the malware family's subgraph that is contained in the sample. A sample is then classified according to the malware family that has the highest similarity score for the sample, as long as this similarity is above a defined *threshold*. If a sample does not meet the similarity threshold for any malware family, then it is classified as cleanware.

A similar approach has been used for classifying a single malware family [50], but multi-class classification using graph mining has not been attempted before. This raises the *research question* ***(RQ4)****: can SCDG-based signatures extracted by symbolic execution be effective in classifying malware?*

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 classifier 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 incorrect classifications for each class independently before appropriately summing them up for computing precision and recall.

In this classification problem, the usual trade-off between sensitivity and false positives comes to light and each vendor chooses their own balance. However in general, it is worth taking into account that mis-classifications between families are less severe than mis-classifications in a binary classifier. Moreover problems could also arise due to the names assigned by different vendors (a very significant problem indeed) since the same malware could be identified with different names and the set of malware belonging to a given (or an aliased) family may not match between vendors. For this reason, we leveraged on VirusTotal for malware identification considering samples that had a large consistency on the attributed family name.

	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

Table 1: Our benchmark of SMT expressions extracted through symbolic executions (timeout of 2 mins per binary).

4. Experimental Method

This section overviews the methodologies used to address the research questions posed in the paper. In the following, we first focus on how we optimized z3 to improve the performance of symbolic execution of malware binaries (Section 4.1), then consider potential heuristics to improve the extraction of traces with Angr tuning (Section 4.2), and finally we describe the optimization of the whole binary analysis toolchain from the symbolic execution to extract the system calls, to the SCDG building process for the classification of binaries according to SCDG-based signatures (Section 4.3).

4.1. Opzimizing z3 for Symbolic Execution of Binaries

To isolate the optimizations of z3 usage in Angr, a separate experiment was done to identify key tactics and tacticals. The experimental design is as follows. A *benchmark of SMT expressions* extracted from symbolic executions of binaries (including both malware and cleanware) is built. Then different z3 tactics are evaluated upon these expressions. The most promising tactics are identified and Angr is modified to use the combination of the identified tactics for the SMT solver in later experiments on the whole analysis toolchain.

To obtain the benchmark SMT expressions we randomly select a subset of the malware and cleanware from the dataset. This includes a mix of malware collected from VirusTotal in 2018 and binaries manually obfuscated with Tigress [27]. De-obfuscation techniques (such as unpacking or de-virtualization) were not considered here since this prevents bias in the benchmark affecting the SMT solver optimization. To prevent overfitting of z3 with the characteristics of the evaluation dataset (see Section 4.3), no binaries were common between the benchmark and evaluation datasets.

Symbolic execution is then attempted on the selected binaries and the expressions sent to z3 are recorded (along with what kind of solution is requested). This creates a benchmark (see Table 1) of the number, kinds, and results of calls from Angr to be used for comparison later. Even a small timeout of 2 minutes for symbolic execution allowed us to collect enough expressions for evaluation.

Collected expressions were solved by z3 with different tactics. As a result of these experiments an *optimized* set of tactics combined by means of tacticals

was chosen to be used in the larger experiments to optimize the whole toolchain. Note that finding an optimized tactic is only a partial answer to **RQ2**, part of the answer also comes from incorporating these results into the analysis toolchain.

4.2. Angr Tuning

There are several tunable parameters in Angr that are fixed in the experiments performed here. The fixed parameters are as follows. The global `timeout` which limits how long Angr can run for before terminating, here fixed to 1 hour. The global `memory` limit which restricts how much RAM Angr can use before stopping the creation of new active states, here fixed to 10GB. The `loop-threshold` built into Angr that stashes a state if an address is visited a certain number of times, here disabled to allow the branching loop threshold introduced below to be effective.

There are also several symbolic execution heuristics in (our customized version of) Angr that can be used to mitigate the limitations of the symbolic execution discussed in Section 2. These are described here along with their values considered in this work.

- `step timeout`: maximum time allowed to compute a step of the symbolic execution. This mainly corresponds to the time consumed by z3 and thus indirectly is related to the choices in **RQ2**.

- `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.

- `branching loop threshold`: a lightweight heuristic to identify the presence of symbolic loops and provide the option to stop unrolling such loops. The difference from the Angr default `loop-threshold` is that the heuristic defined here only considers visits involving loops with symbolic variables in their condition, while Angr's default `loop-threshold` simply takes into account the repeated visit to the same address in the program's address space disregarding the participation of symbolic or concrete variables. Observe that Angr's approach acts "blindly" and does not take into account that only symbolic loops cause a complexity increase (due to the creation of new states, see Section 2), thus the motivation for the optimized heuristic here. Note that this heuristic addresses **RQ1** by possibly improving the code exploration.

Note that there are many techniques for making an informed guess when no useful information could be extracted from the loop such as the Loop-Extended Symbolic Execution (LESE) [66], the Read-Write set (RWset) [67] and the bit-precise symbolic loop mapping [68]. 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. For example, in case of `for` loops with a conditional statement executed symbolically, new execution branches can possibly be spawned until the condition is not meet whereas the "for body" modifies symbolic variables. This scenario could easily create a state-space explosion, or generally high resource consumption. Our heuristic detects the presence of this situation (since at the end of the "for body" the program counter points again to the "for header") and therefore chooses to momentarily stop the loop unrolling, and possibly come back on it in case enough resources are left.

The reason for having fixed the (global) `timeout` and `memory` parameters lies in our choice to do not add any constraint to the exploitable resources (modulo the resources available in our experimental configuration). The `loop-threshold` instead has not been investigated since our `branching loop threshold` is more sophisticated and can be considered as inclusive of the former.

4.3. Optimizing Binaries Classification Based on the SCDG

Our experiments aim at: (i) pinpointing the graph metrics corresponding to better SCDGs for malware classification to address **RQ3**; (ii) evaluating the accuracy of the classification by F-score to address **RQ4**, and (iii) explaining how heuristics affect the quality of the traces extracted by means of symbolic execution and the graph building process. This last (iii) address a further *research question **(RQ5)**: examining the relations between the tuning and evaluation.*

The experiments on the whole analysis toolchain involve extraction of execution traces from binaries with Angr, construction of SCDGs and their classification. Therefore we evaluate (i) the symbolic execution heuristics (see Section 4.2), (ii) customized set of z3 tactics identified by the experiment defined in Section 4.1 (results are in Section 5.1), and (iii) the graph building heuristics (see Section 3.1). For the heuristics we used a *full factorial design* allowing us to identify their effect. For each factor we have considered two levels according to our past experience. The list of factors and their levels are summarized in Table 2. *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 taking into account that the number of experimental units grows as number of levels to the power of the number of factors (128 units in our case).

To assess the classifier's ability in generalizing the predicted class (i.e., either malware including its family, or cleanware) for binary samples we used *5-fold cross-validation*. This technique allows us to reduce dependency from the dataset used during the learning phase and thus avoid overfitting.

Experiments have been performed on a cluster constituted by 12 machines having four Intel CPU E5-2660 v4 @ 2.00 GHz with 132 GB of RAM each, running Ubuntu 16.04 LTS, Angr build 7.8.8.1 and z3 4.5.1.0.post2. The evaluation dataset is composed of 25 binaries for each of 8 malware families collected in 2018 (*couponmarvel*, *gamemodding*, *installbrain*, *multiplug*, *jeefo*, *detroie*, *mira* and *addrop*), and 25 cleanware binaries (taken from a clean machine running

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

Table 2: Levels for each factor of the graph building and symbolic execution heuristics

	`simplify()`		`satisfiable()`		`batch_eval()`		`max()`		`min()`	
Tactic	# solved	Exec. time	# solved	Exec. time	# solved	Exec. time	# solved	Exec. time	# 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**
Total expressions	54460		103881		47585		177439		112712	

Table 3: Results of the selected z3 tactics over our benchmark of extracted SMT expressions. Exec. time is a total time for solving all expressions in the benchmark. The performance of the best tactic for each request type is in bold.

Windows 7 Pro). SHA1 and family of each binary are reported in Appendix A.5. Cleanware signatures are *not* used in the learning phase since cleanware does not have a common behavior. Note that a novel malware would likely be classified as cleanware since its behavior is unknown to the classifier, however this is inherent to signature-based detection.

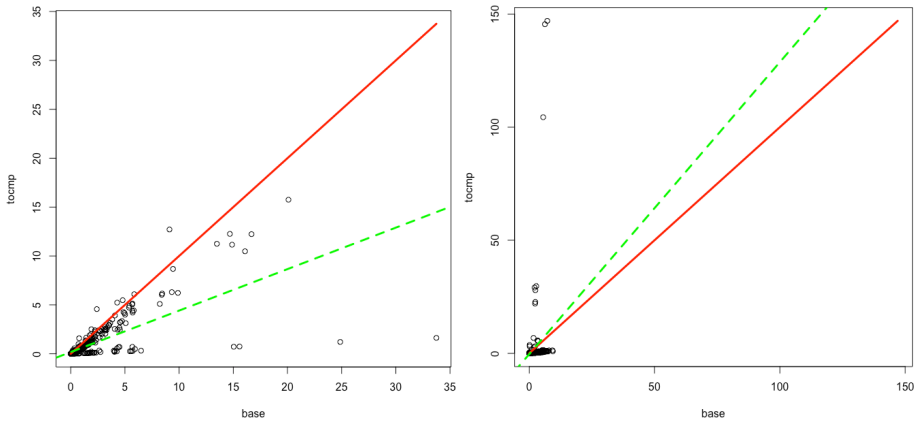
Results of our comparative experiments are analyzed considering the correlations between each possible pair of variables (i.e., graph metrics, heuristics and response variables) and according to the *Analysis of Variance (ANOVA)*.

5. Results, Analysis and Discussion

This section discusses the results of the experiments to address the RQs. The first section explores optimizing usage of the z3 SMT theorem solver to generate a new "optimized strategy". The second section details the results for the whole analysis toolchain and variation of all parameters.

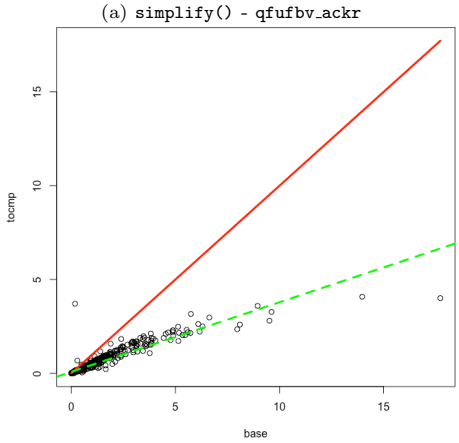
5.1. Optimizing Usage of z3

To create an "optimized strategy" for use on the whole analysis toolchain experiments, an evaluation of z3's behavior on typical Angr calls was performed. The performance of z3 built-in tactics (including `NO_TACTIC`) on all the benchmark expressions was evaluated. Initially no limits were put on the resources available for solving the expressions and this led to some expressions requiring

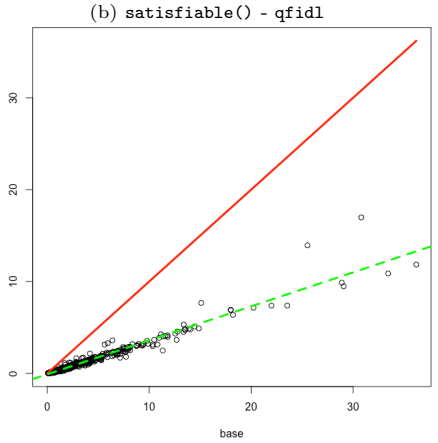


(a) `simplify() - qfufbv_ackr`

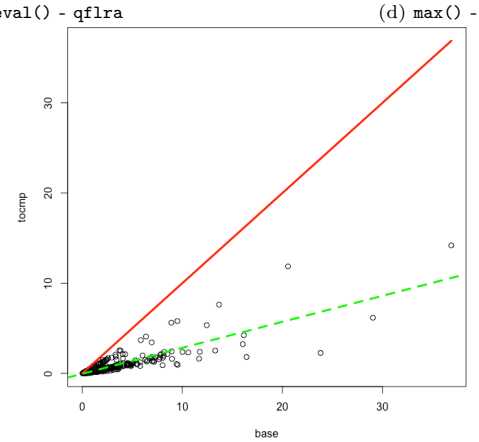
(b) `satisfiable() - qfidl`



(c) `batch_eval() - qflra`



(d) `max() - qfauflia`



(e) `min() - smt`

Figure 3: 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 function in a given binary. On the x-axis the `NO.TACTIC`, on the y-axis the compared tactic. The dashed line in light green represents a linear regression model. If the green line is below the diagonal (in red), the customized tactic outperforms the default one in terms of execution time.

several hours for z3 to find definitive results with a given tactic. To achieve efficient symbolic execution in practice and resolve this potential time cost, a timeout was set for z3 of one minute. `unknown` is returned when the timeout is reached without a definitive answer (`sat` or `unsat`). (Note that one minute was chosen to allow sufficient time for z3 to attempt to solve, while also not causing prohibitive time cost in symbolic execution where many calls to z3 are made for each binary, as in the experiments with the whole analysis toolchain.) The evaluation of each tactic against each SMT expression was repeated 30 times and the results averaged to prevent concerns from caching, scheduling, OS behavior, etc.

Two metrics have been considered in evaluating the effectiveness of tactics: execution time and ability to successfully perform a satisfiability check. The baseline considered here is the `NO.TACTIC` tactic that does not exploit any special knowledge about the SMT expression to be solved and without optimizing SMT solver usage. A summary of the performance of the selected tactics over the benchmark of SMT expressions is presented in Table 3. The best result for each type of call is remarked in bold, note that this is the least time spent, potentially including the timeout in case of failure to find a solution.

For each of the winning tactics from Table 3, a deeper analysis is done that compares all the calls of each type (e.g., `simplify`) from a single binary. This is done to prevent over-fitting of the tactic to expression type. Results are presented in Fig. 3 where each plot compares the base time taken per binary (x-axis) with the time taken using the best tactic for that expression type (y-axis) as chosen from Table 3.

Fig. 3a compares `NO.TACTIC` and `qfufbv_ackr` (Quantified Free formulas (QF) over bitvectors (BV) with uninterpreted sort function (UF) and symbols solved with Ackermanization) on the execution time to solve the SMT expressions from `simplify()` in the benchmark. Observe that (with the exception of a very few binaries) the benefit of using the `qfufbv_ackr` in place of the default tactic is clear.

As reported in the scatter plot in Fig. 3b the `qfidl` (Boolean combinations of inequality constituted by differences between integers variables and constants) did not produce significant advantage when considering the whole binary file over the default tactic for the `satisfiable()` calls.

By contrast, the expressions from `batch_eval()` using 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 except for a very few (see Fig. 3c).

From Table 3 it is possible to observe that `max()` and `min()` are the most common requests and also account for most of the time spent by the solver. By selecting the right tactic, respectively `qfauflia` (closed QF linear formulas over the theory of integer arrays extended with free sort and function symbols) and `smt` (Boolean SAT-based SMT solver), the default configuration could be outperformed for *all* the binaries as shown in the scatter plots in Fig. 3d and 3e.

Results in Table 3 and Fig. 3 confirm that heuristic proof strategies are rarely "one size fits all" [62]. Thus, we have created a new "optimized" strategy based

on the best tactics on the benchmark (`qfufbv_ackr`, `qfidl`, `qflra`, `qfauflia`, and `smt`) combined with parallel tacticals. (Note that while `qfidl` did not produce significant advantage on binaries for `satisfy()` as shown in Fig. 3b, we chose to still include it as there were cases with relative advantage, also in the other kinds of calls.) This "optimized strategy" is compared with the default one in the overall analysis toolchain results below.

The above addresses part of **RQ2**, in that there are clear optimizations in the usage of the z3 SMT theorem prover by changing the tactics.

5.2. Overall Toolchain Optimization

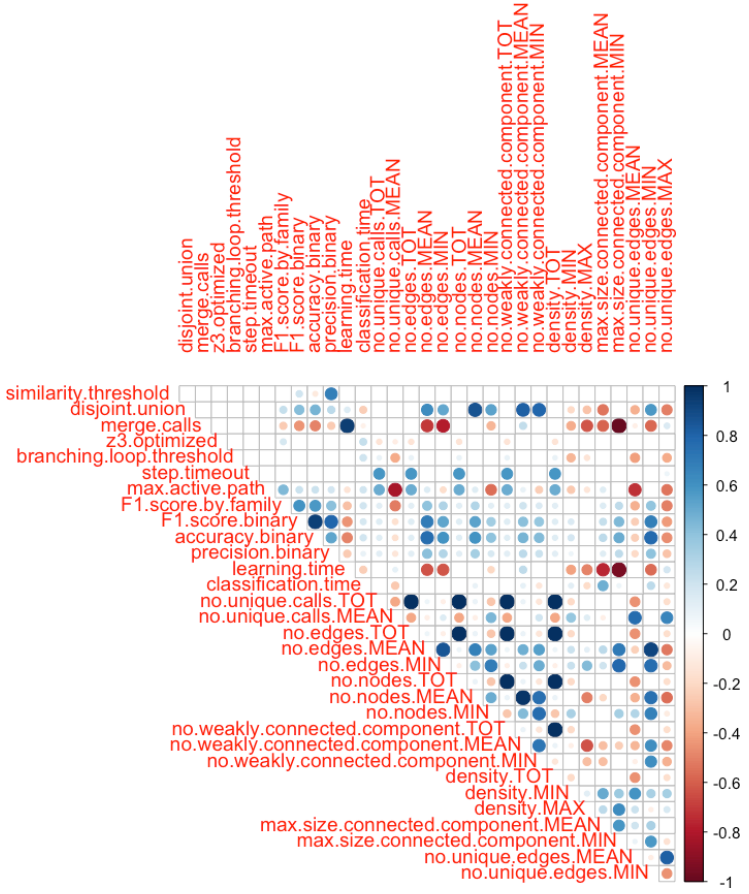


Figure 4: Correlation matrix (p-*value* < 0.01) for the graph building and symbolic execution heuristics, graph metrics and performance of the classifier (some correlations have been removed from the plot for the sake of clarity).

As discussed in Section 4 we use the F-score to evaluate the analysis toolchain results.

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 presented in Fig. 4. For the graph metrics described in Section 3.1 we computed some summary statistics (total, mean, standard deviation, min, quartiles Q_1, Q_2, Q_3 and max). Only correlations with significance level greater than 0.01 are considered here. 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, and also learning time. 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 3.1). The negative correlation for learning time is due to gSpan being much slower on large graphs.

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 3.1) 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 a higher number of sub-graphs and thus requires a higher learning time. On the other hand, the presence of a 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 Fig. 4).

The main correlations of interest to the RQs are as follows. Regarding **RQ1** the branching loop threshold allows us to collect more unique calls and edges but this is not reflected in the F-score (that mainly depends on the connected components as already observed). Also, not limiting the max active paths is positively correlated with the F-score thanks to its ability to explore the CFG in a BFS order. Regarding **RQ2** our "optimized" z3 tactic has a positive correlation with the F-score (as later exploited in Section 5.3).

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 the presence and size of connected components. This is an important and unexpected finding because contrarily to what one can expect the cornerstone is not the number of (unique) calls that reflect "how deep" the code is explored. 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 3.2). Here it is important to note that the properties of the connected components are not parameters under direct control but rather the result of heuristics and techniques used during the analysis toolchain and also influenced by the behav-

ior of the binary itself. Even if it is not possible to identify which is the "best" size for the connected component, Appendix A.3 explores size and number of the connected components and their relationship with the performance of the classifier.

5.3. 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 compactness, Fig. 5 shows only three out of the seven studied factors (for the sake of completeness the impact of all the factors is reported in Appendix A.1).

Regarding **RQ2** the "optimized" tactic for z3 allows the consolidation of the classifier to an F-score above 0.825, whereas the default version has widely spread performance even reaching a very low F-score of 0.65 (see Fig. 5c).

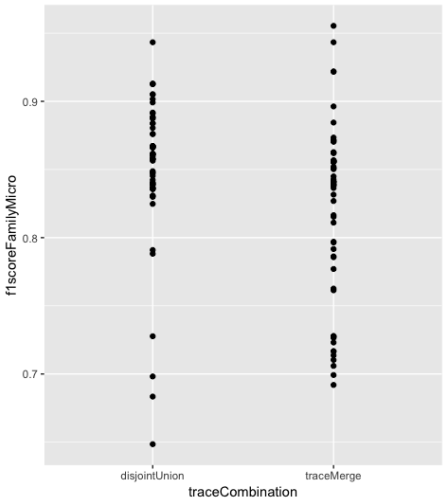
To address **RQ3** for the graph building heuristics, performing the disjoint union of the traces is generally preferable (see Fig. 5a), as well as to not merge the calls (not shown in the figure). Instead, using a limit to the minimum trace size to build a graph does not have a direct impact on the classifier. This happens because when Angr is able to analyze the malware (see the following Section 5.5), generally it is able to extract a substantial number of calls.

Limiting the max active paths results in very inconsistent performance of an F-score from 0.65 to 0.94. Finally, branching loop threshold and step timeout show a similar behavior where, if disabled, the performance are more constant.

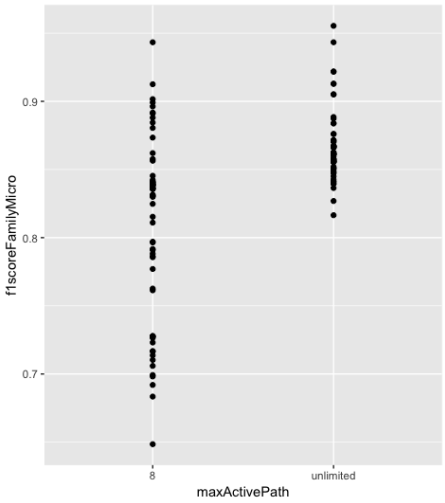
The above overview addresses **RQ5** where clearly there are significant relations between the binary exploration behavior, behavioral signatures, and classification outcomes.

5.4. Classification Accuracy and Final Remarks

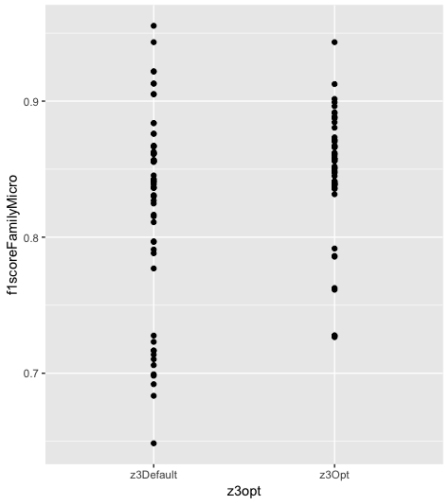
In this section we discuss the results on the whole toolchain. During our experiments, the best classifier scored an F-score by malware family of 0.955 and F-score for the binary classifier of 0.970 (according to our preliminary analysis and experiences we set a similarity threshold for gSpan of 0.7, further results supporting this are reported in Appendix A.4). This addresses **RQ4** by showing that the multi-class classification built on SCDG-based signatures extracted by symbolic execution is effective, albeit somewhat sensitive to the choices of other parameters in the experiments. Note that the results here should not be considered a *direct* comparison with related works (see also Section 6). Our focus is the optimization of the symbolic execution and the heuristics evaluation, as already described in Section 4, the adoption of a full factorial design approach better suits our needs but is very computationally demanding. As a result, on first glance our dataset may appear small (in terms of both number of families and samples). Observe that symbolic execution trades more execution time for significantly more complete results, particularly on complex samples such as malware. Since we conducted full factorial experiments over 128 units and 5-fold cross validation, this yields 640 classification experiments each over 200 samples. In practice this makes the experiments here quite exhaustive, indeed



(a) Trace combination



(b) `max active path`



(c) `z3 config`

Figure 5: Main effects of the factors over the F-score. Levels for a given factor on the x-axis, and the F-score on the y-axis.

adding just 25 more samples (or one family according to the the family size in our dataset) would have increased experimentation time by more than five days. Thus, although the dataset may appear small, in practice this dataset is larger than many works that consider symbolic execution that perform analysis on dataset composed of a couple of dozen to a maximum of a few hundred samples [69, 45, 60, 70, 71].

Although results on a larger dataset would be ideal, we note that using such intensive techniques on all samples is largely an academic practice. *In practice, both symbolic and dynamic execution are applied only to a subset of the collected malware samples when other less demanding techniques have not been able to provide enough confidence on the results* [72].

To further complicate direct comparison, the community lack a common public database with a large variety of well labeled malware [73]. Finally, although the community more often uses the F-score even in case of multi-class classifier, here we prefer micro-average F-score (described in Section 3) due to its ability to better cope with classes and provide more precise results in presence of imbalance.

We adopted 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 (reported in Figures A.8, A.9 and A.10 of Appendix A.2). Trace merge combined with merge calls is too aggressive in simplifying the graphs yielding a 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 5.2. Even though 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. On the other hand, the interaction plot of merge calls and step timeout shows independence of these factors. A very interesting dependency exists 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 is tolerable), and also to make the symbolic execution a little closer to 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.

We finally considered the interactions between each possible combination of factors. The analysis shows several interaction effects asserted with a signifi-

cance lower than 0.001. The Pareto chart (not shown here for the sake of brevity, but included in Appendix A.2) 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 the following.

- Merge calls is very "risky", even if it is used in several of the best configurations. Similarly, disjoint union has more constant performance than trace merge.
- 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 effects principle* (stating that the system is dominated by the effect of the main factors and low-order-factor interactions) does not hold.

In this section we answered to three important questions: "which are the most critical SCDG metrics?" (**RQ3**); "can SCDG-based behavioral signatures be used in an effective multi-class malware classifier?" (**RQ4**); and "how the heuristics are interrelated and what is their impact on the malware classification?" (**RQ5**).

5.5. Threats to Validity

This section discusses the validity [74] of our study according to: construct, internal, external validity and reliability.

Construct Validity: we have performed controlled experiments that allowed us to tune the environment and measure the metrics of interest. In this regard, no issue should arise from our experiential study. We remark that the presented

exploratory study concerns more the exposing of which factors are most likely to influence the outcome than on setting up a quantitative model.

Internal Validity: 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. This design choice has strictly impacted the external validity (discussed in the following). Indeed, by adopting the full factorial experiment design the number of experiments corresponds to $\#levels^{\#factor}$, where each experiment required us about 3.5-4 hours. Therefore adding a single factor would have requested us to perform 192 experiments instead of 128 and thus an additional 1.5 week of computation, and to add two more families about 1 hour for each experiment. Notwithstanding, our experiment design with two levels per factor (as usually adopted in such a methodology), provides sounder results with respect to a fractional factorial design (the adoption of a response surface methodology is not feasible for timing reason). The choice of a full factorial design in place of a, computationally lighter, fractional one turned out to be wiser since (as noted above) the sparsity of effects principle does not hold.

External Validity & Reliability: The choice of Angr as our *symbolic execution framework* should not affect the validity since the heuristics considered here could be implemented in other frameworks with some engineering effort (e.g., writing plugins for S^2E or Triton). The *datasets* used are a common concern in malware research. Our dataset was limited to binaries in PE format and considering 8 known malware families (and cleanware) collected in 2018 (that we have not extended for experiment timing reasons as discussed above). However, like most other works we cannot make this dataset public (file hash and family are reported in Appendix A.5), and the labeling by antivirus is a challenge. Our dataset considered samples where a very strong consensus was shown by the antivirus engines on *VirusTotal* and the construction of a similar dataset should be straightforward. Another area is to *consider the effect of packed binaries* since the symbolic execution framework could spend time analyzing the packer (instead of the packed binary payload) and therefore extract 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 limited our exploration to the packer, the rest of the analysis has not been affected. Lastly, general limitations of symbolic execution frameworks could limit the applicability of the considered malware classification approach. Angr has a limited set of models for system calls and only for specific versions of libraries. Lack of information for other system calls or incorrect models due to changes in different versions of the library could prevent a complete and correct analysis. The same limits are also present in cases when particular input parameters or network resources are required to start the execution.

Assuming the use of a similar set of malware families (and a family classification consistent with ours), having adopted the ANOVA test and k-fold validation obtaining evidence with a low significance level (0.001), our findings

should be congruent with ones hypothetically drawn by other researchers (and thus *reliable*).

6. Related Work

With the aim of helping to understand the context of this paper, this section overviews some related works adopting techniques different from ours but considering a similar domain/setting. Note that since this work explores the optimization of an entire toolchain, the related works only relate to individual parts/aspects of this paper. The organization of this section attempts to align the related works to the components optimized in this paper.

6.1. Symbolic Execution

Symbolic execution is a common way to try and analyze the behavior of a program [75, 76]. This can apply to programs in many forms including source code or binary [75, 76]. Since this work explores binary analysis, the rest of this section considers only binary analysis with symbolic execution.

There are several available tools that can perform binary analysis using symbolic execution [58, 17, 18, 19]. Although this work uses only Angr [58] for the experiments, the concepts apply to tools including Mayhem [17], S^2E [18] and Triton [19] amongst others. In practice, each tool has its own optimizations and weaknesses, as well as its own ability to be (re)configured to modify its execution choices. Since the goal here is to consider a holistic approach to the whole toolchain, the results can be adapted to any of these tools (or others), albeit with some experimentation to consider peculiarities and advantages each tool brings.

We observe that our modifications to the symbolic execution behavior here can be easily adapted to many other tools, particularly since we focused on lightweight modifications that can yield significant improvements considering the other components and algorithms in our (or similar) toolchain. We believe the heuristics used here can be used as a guide to other symbolic execution experiments.

There is also some related work in improving symbolic execution for specific domains [77, 78, 79, 80]. In [77] the authors consider how to steer the symbolic execution engine to specific paths of interest. Its bottom line idea lies on considering execution sub-traces (constituted by n conditional branches) and using statistical analysis of the already covered sub-traces to determine which execution states may steer towards the less explored paths. There is also research into optimizing the SMT solver used in a symbolic execution engine [78, 79], although the focus is on reducing the input to the SMT solver. In [80] the authors consider the merging of states, which has some conceptual similarities to our *traces-merge* (described in Section 3.1) even if in their case the merging happens during the execution itself reducing the introduction of auxiliary symbolic variables.

6.1.1. SMT Optimization

One aspect of this work is in finding optimizations for the SMT solver that is used in the symbolic execution engine [78, 79]. SMT optimization has been studied before [63, 64, 78, 79], however these works focus on solving a particular problem that is known a priori. In [63] the authors are optimizing their SMT solver for quantified bitvector operations. Similarly, in [64] the optimization is for linear integer/real arithmetics. Although [78, 79] are focused explicitly on symbolic execution with SMT optimization, the problems are again defined a priori rather than discovered during exploration.

However, this work approaches SMT optimization from a very different perspective. Since there is no way to know a priori what expressions need to be optimized, the approach used here is the only reasonable way to attempt efficient optimization. In particular, by gathering a suitable sample set and discovering which expressions are most expensive, and which approaches can yield the best performance improvements.

The above said, we welcome the use of the results of this work to guide future SMT optimizations and to consider which approaches are most significant for this problem. Since we have only exploited available tacticals, we recognize that the SMT optimization community may be able to use the approach here to significantly optimize an SMT solver for this exact problem in ways that go beyond mere tactical choices.

6.1.2. Binary Exploration Heuristics

Several sophisticated loop un-rolling 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) [66], the Read-Write set (RWset) [67] and the bit-precise symbolic loop mapping [68]. LESE [66] 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 [67] 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 [68] aims at identifying the semantics of possible cryptographic algorithms used in obfuscated binary code.

Differently from these techniques, being a pure heuristics our approach turns out to be much less computational demanding.

6.2. Malware Classification

Malware classification is a widely explored area with many different approaches. In the present work, we focused on malware classification using SCDGs as behavioral signatures as is common to [45, 46, 47, 48, 49, 51, 52, 50]. Since the goal of this work is on the toolchain optimization to achieve improvements in malware classification, and the availability of standard datasets is very

poor, a direct comparison of results with other works is difficult. However, this section offers some brief comments on those most closely related to this work.

To the best of our knowledge, only a few recent works have tried to classify PE (Windows) malware by their family and none report classification results by using symbolic execution. By extracting static features (i.e., textual and binary patterns) with YARA, Sun et al. [81] achieved an F score of 0.936 with a random forest classifier (RF) over a dataset constituted by 12 families. Still using a RF but extracting behavioral traces through a concrete execution within the Cuckoo sandbox, Hansen et al. [82] achieved an F-score of 0.864 on a dataset of 5 malware families. Thanks to an RF fed with a mix of static and dynamic features obtained by executing and monitoring the execution of binaries with the HookMe tool [83], Islam et al. [84] achieved a trueness of 0.97 (misclassifications are not reported). A behavioral graph representation similar to the SCDG but extracted by means of taint analysis and dynamic execution in a sandbox is adopted also by Ding et al. [85]. There, authors remove duplicate subgraphs representing identical call sequences and group all the subgraphs extracted from instances of the same malware family. The graph matching is then performed by isomorphism. Experiments on a dataset of 6 families with about 200 samples each show an accuracy of about 0.96 when doing binary classification between a given family and cleanware (no multi-class classification is performed, but six independent binary classifiers are used for the results, one per family).

In [86] the authors consider dependency graphs similar to our SCDGs and use graph similarity (although not common subgraphs directly) to classify malware. Their approach is to create clusters (with a priori knowledge of how many clusters to create) of graphs by their distance and then compare these clusters with ground truth as a form of classification. When tested on a dataset of 300 samples from 6 families their results are mixed with some families achieving only $\sim 50\%$ accuracy. By contrast, the work here does not need to know how many clusters/families to attempt to group the samples into, instead using signatures from known samples as is typical on malware classification.

In [50] the authors use a similar common subgraph approach to the algorithm adopted here, although only for binary classification (a single family and the rest being clean). Their results achieved very high accuracy (they targeted $F_{0.5}$-score and achieved 99.40%) with tuned parameters to their algorithms. However, this was only for a single family and not multi-class classification as in this work. That said, the optimizations here would apply directly to their approach.

Note that other works have similar approaches in concept [87, 86, 88, 89]. These include using other forms of behavior to SCDGs [87, 86, 88, 89] and some form of graph similarity for clustering of common behaviors [86, 88, 89].

In [87] the authors use SVM (Support Vector Machine) on a vector of logged behaviors from running the malware on a sandboxed environment. They claim to improve results on malware not already detected by some other system by approximately 70%, but do not give detailed results. Their similarity to our work is that they detect malware families using some kind of behavioral signatures and machine learning techniques.

In [88] the authors use clustering of call graphs to infer malware samples.

They use a dataset of 194 samples from 24 families and conclude that the *k-medoid* clustering (i.e., a partitioning algorithm similar to the well-known k-means but in which the centers of the clusters are chosen among the data points) is not effective. The density-based cluster algorithm DBSCAN, not requiring any prior knowledge on the number of cluster, showed instead better performance. Therefore, DBSCAN is evaluated over two larger datasets respectively of 675 and 1050 samples. Results prove that the clusters are correctly and wrongly identified respectively for 29 and 19 clusters, and for 36 and 14 in the second dataset. For both the datasets there are a good amount of samples not classified in any cluster (respectively 260 and 253). In contrast to our work, authors here adopt an unsupervised classifier, mainly focusing on keeping clusters homogeneous (i.e., with all the samples of a cluster belonging to a single family) even if their number should prove to be significantly higher than in the ground truth provided in input.

Graph-based representation of malicious behaviors are adopted even for malware running on the Android platform. The samples are generally executed dynamically but similar classifiers are used to the ones adopted in the Windows domain, e.g., SVM [90], graph isomorphism [91], Markov Chain Monte Carlo sampling [92], graph kernel [93] and Convolution Neural Network [94]. Even if the absolute performance results of the classifiers can not be compared with our work, given the presence of several similarities with our behavioral analysis toolchain, many of the optimizations and lessons learned here would apply to their works.

7. Conclusion

The tools used to determine software correctness, reliability, and security rely upon many formal and heuristic configurations. The work here explored the role of these in the scenario of malware program analysis, chosen as a particularly challenging form of software analysis. The results addressed five research questions, overall giving deeper insight into the usage and configuration of analysis tools.

The results here showed: (i) how to prioritize state exploration in symbolic execution; (ii) how to tune an SMT solver for symbolic execution; (iii) how to effectively construct SCDG-based behavioral signatures of programs; (iv) that a multi-class behavioral machine learning built on SCDG-based signatures extracted by symbolic execution is effective for malware classification; and (v) that there are many positive and negative correlations between heuristics. In particular, the above shows that significant improvements can be made in analysis engines and toolchains, and that components cannot be optimized in isolation – the whole toolchain must be considered.

Future Work: It would be interesting to contrast the proposed optimizations into other tools exploiting similar combinations of formal and heuristic approaches for software analysis [1, 2, 3, 4, 95, 5, 13] with the purpose of understanding both: the presence of domain-specific results, and evaluating the opportunity to generalize these results.

Moreover, since our toolchain uses malware classification based on graph similarity, the graph matching process takes on great importance. In particular, an implementation of *approximate subgraph matching* could yield interesting results. Different techniques from the graph edit distance adopted in this paper have been proposed and it would be interesting to evaluate their impact in our malware analysis toolchain. Examples include: Markov Chain Monte Carlo sampling technique [96], rewriting the adjacency matrix representing the graph as a least-square problem [97], or Graph Convolutional Networks [98].

Finally, it would be also interesting to evaluate the performance of our optimization in case of malware analysis toolchains relying on graph clustering (i.e., based on unsupervised machine learning). For example, in the domain of security Toffalini et al. [89] adopt the Markov Chain Cluster (MCL) method to identify malicious activities as anomalies with respect to regular user events, whereas Kinable and Kostakis [88] cluster call graphs with a density based algorithm (DBSCAN) on graph pairwise distance.

Acknowledgements

The authors would like to thank *VirusTotal.com* for providing access to their API and Cisco for the malware samples used in the experiments.

References

[1] P. Cousot, R. Cousot, J. Feret, A. Mine, L. Mauborgne, D. Monniaux, X. Rival, Varieties of static analyzers: A comparison with astree, in: IEEE/IFIP TASE '07, 2007, pp. 3–20.

[2] MathWorks. Polyspace [online] (March 2019). Accessed: May 20, 2019.

[3] M. Bradley, F. Cassez, A. Fehnker, T. Given-Wilson, R. Huuck, M. Junker, GoannaSMT — a static analyzer with SMT-based refinement, in: Tools for Automatic Program AnalysiS (TAPAS 2012), ENTCS, Deauville, France, 2012.

[4] S. Bardin, R. David, J.-Y. Marion, Backward-bounded DSE: Targeting infeasibility questions on obfuscated codes, in: 2017 IEEE Symposium on Security and Privacy (SP), IEEE, 2017. doi:10.1109/sp.2017.36.
URL https://doi.org/10.1109%2Fsp.2017.36

[5] A. Johnsen, K. Lundqvist, K. Hänninen, P. Pettersson, M. Torelm, Experience report: Evaluating fault detection effectiveness and resource efficiency of the architecture quality assurance framework and tool, in: 28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017, 2017, pp. 271–281. doi:10.1109/ISSRE.2017.31.
URL https://doi.org/10.1109/ISSRE.2017.31

[6] Clang. Clang documentation [online] (March 2019). Accessed: May 20, 2019.

[7] S. C. Chan, G. R. Gao, B. Chapman, T. Linthicum, A. Dasgupta, Open64 compiler infrastructure for emerging multicore/manycore architecture all symposium tutorial, in: IPDPS 2008, 2008, pp. 1–1.

[8] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, T. Waterman, Acme: Adaptive compilation made efficient, SIGPLAN Not. 40 (7) (2005) 69–77. doi:10.1145/1070891.1065921.

[9] M. Junker, R. Huuck, A. Fehnker, A. Knapp, Smt-based false positive elimination in static program analysis, in: ICFEM 2012, 2012.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, SIGPLAN Not. 40 (6) (Jun. 2005). doi:10.1145/1064978.1065034.

[11] C. Calcagno, D. Distefano, Infer: An automatic program verifier for memory safety of c programs, in: NASA Formal Methods, 2011.

[12] A. M. Braga, R. Dahab, N. Antunes, N. Laranjeiro, M. Vieira, Practical evaluation of static analysis tools for cryptography: Benchmarking method and case study, in: 28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017, 2017, pp. 170–181. doi:10.1109/ISSRE.2017.27.
URL https://doi.org/10.1109/ISSRE.2017.27

[13] A. Aquino, G. Denaro, P. Salza, Worst-case execution time testing via evolutionary symbolic execution, in: 29th IEEE International Symposium on Software Reliability Engineering, ISSRE 2018, Memphis, TN, USA, October 15-18, 2018, 2018, pp. 76–87. doi:10.1109/ISSRE.2018.00019.
URL https://doi.org/10.1109/ISSRE.2018.00019

[14] Radare2 Team, Radare2 Book, GitHub, 2017.

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

[16] Binary Ninja. Malware analysis and automation with binary ninja [online] (June 2018). Accessed: May 20, 2019.

[17] S. K. Cha, T. Avgerinos, A. Rebert, D. Brumley, Unleashing mayhem on binary code, in: SP '12, 2012.

[18] V. Chipounov, V. Kuznetsov, G. Candea, S2e: A platform for in-vivo multi-path analysis of software systems, SIGPLAN Not. 47 (4) (2011) 265–278.

[19] F. Saudel, J. Salwan, Triton: A dynamic symbolic execution framework, in: Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, 2015.

[20] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, A. Pretschner, Code obfuscation against symbolic execution attacks, in: ACSAC '16, 2016.

[21] N. Kuzurin, A. Shokurov, N. Varnovsky, V. Zakharov, On the concept of software obfuscation in computer security, in: ISC 2007, 2007.

[22] Z. Wang, J. Ming, C. Jia, D. Gao, Linear obfuscation to combat symbolic execution, in: ESORICS'11, 2011.

[23] C. Collberg, S. Martin, J. Myers, J. Nagra, Distributed application tamper detection via continuous software updates, in: ACSAC '12, 2012.

[24] Oreans Technologies. Themida: Advanced windows software protection system [online] (February 2017). Accessed: May 17, 2019.

[25] VMProtect Software. Vmprotect - new-generation software protection [online] (May 2018). Accessed: May 17, 2019.

[26] Oreans Technologies. Code virtualizer: Total obfuscation against reverse engineering [online] (February 2017). Accessed: May 17, 2019.

[27] C. Collberg. The Tigress C Diversifier/Obfuscator [online] (2012). Accessed: May 17, 2018.

[28] M. Yusuf, A. El-Mahdy, E. Rohou, On-stack replacement to improve jit-based obfuscation a preliminary study, in: JEC-ECC 2013, 2013.

[29] J. Bellizzi, M. Vella, Wexpose: Towards on-line dynamic analysis of web attack payloads using just-in-time binary modification, in: ICETE 2015, 2015.

[30] A. Kalysch, J. Götzfried, T. Müller, Vmattack: Deobfuscating virtualization-based packed binaries, in: ARES '17, 2017.

[31] S. Naval, V. Laxmi, M. S. Gaur, P. Vinod, Spade: Signature based packer detection, in: SecurIT '12, 2012.

[32] W. Yan, Z. Zhang, N. Ansari, Revealing packed malware, SP '08 (2008).

[33] U. Bayer, E. Kirda, C. Kruegel, Improving the efficiency of dynamic malware analysis, in: SAC '10, 2010.

[34] C. K. Patanaik, F. A. Barbhuiya, S. Nandi, Obfuscated malware detection using api call dependency, in: SecurIT '12, 2012.

[35] Z. Wu, S. Gianvecchio, M. Xie, H. Wang, Mimimorphism: A new approach to binary code obfuscation, in: CCS '10, 2010.

[36] H. Agrawal, L. Bahler, J. Micallef, S. Snyder, A. Virodov, Detection of global, metamorphic malware variants using control and data flow analysis, in: MILCOM 2012, 2012.

[37] S. Alam, R. N. Horspool, I. Traore, Mard: A framework for metamorphic malware analysis and real-time detection, in: 2014 IEEE 28th International Conference on Advanced Information Networking and Applications, 2014, pp. 480–489.

[38] Z. Bazrafshan, A. Hamzeh, Metamorphic malware categorization using co-evolutionary algorithm, in: 2015 7th Conference on Information and Knowledge Technology (IKT), 2015, pp. 1–6.

[39] M. R. Chouchane, A. Lakhotia, Using engine signature to detect metamorphic malware, in: Proceedings of the 4th ACM Workshop on Recurring Malcode, WORM '06, ACM, New York, NY, USA, 2006, pp. 73–78.

[40] M. Egele, T. Scholte, E. Kirda, C. Kruegel, A survey on automated dynamic malware-analysis techniques and tools, ACM Comput. Surv. 44 (2) (2008) 6:1–6:42.

[41] C. Willems, T. Holz, F. Freiling, Toward automated dynamic malware analysis using cwsandbox, IEEE Security Privacy 5 (2) (2007) 32–39.

[42] A. Bulazel, B. Yener, A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web, in: ROOTS 2017, 2017.

[43] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, I. Finocchi, A survey of symbolic execution techniques, CoRR abs/1610.00502 (2016). `arXiv:1610.00502`.

[44] S. Aboughadareh, C. Csallner, M. Azarmi, Mixed-mode malware and its analysis, in: PPREW-4, 2014.

[45] M. Christodorescu, S. Jha, C. Kruegel, Mining specifications of malicious behavior, in: ESEC-FSE '07, 2007.

[46] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, X. Yan, Synthesizing near-optimal malware specifications from suspicious behaviors, in: SP '10, 2010.

[47] H. D. Macedo, T. Touili, Mining malware specifications through static reachability analysis, in: ESORICS 2013, 2013.

[48] F. Song, T. Touili, Pushdown model checking for malware detection, in: TACAS 2012, 2012.

[49] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, E. Kirda, A quantitative study of accuracy in system call-based malware detection, in: ISSTA 2012, 2012.

[50] N. B. Said, F. Biondi, V. Bontchev, O. Decourbe, T. Given-Wilson, A. Legay, J. Quilbeuf, Detection of mirai by syntactic and behavioral analysis, in: 29th IEEE International Symposium on Software Reliability Engineering, ISSRE 2018, Memphis, TN, USA, October 15-18, 2018, 2018, pp. 224–235. doi:10.1109/ISSRE.2018.00032.
URL https://doi.org/10.1109/ISSRE.2018.00032

[51] K. H. T. Dam, T. Touili, Automatic extraction of malicious behaviors, in: MALWARE 2016, 2016.

[52] M. Dimjašević, S. Atzeni, I. Ugrina, Z. Rakamaric, Evaluation of android malware detection based on system calls, in: IWSPA '16, 2016.

[53] M. Fan, J. Liu, X. Luo, K. Chen, T. Chen, Z. Tian, X. Zhang, Q. Zheng, T. Liu, Frequent subgraph based familial classification of android malware, in: ISSRE 2016, 2016, pp. 24–35.

[54] VirusTotal. YARA: The pattern matching swiss knife for malware researchers (and everyone else) [online] (2019). Accessed: Dec 3, 2019.

[55] EclecticIQ. What is STIX and TAXII? The industry standards for Cyber Threat Intelligence [online] (December 2019). Accessed: Dec 15, 2019.

[56] National High Tech Crime Unit of the Netherlands' police, Europol's European Cybercrime Centre, Kaspersky Lab, McAfee. No More Ransom [online] (December 2019). Accessed: Dec 15, 2019.

[57] E. M. Rudd, F. N. Ducau, C. Wild, K. Berlin, R. E. Harang, ALOHA: auxiliary loss optimization for hypothesis augmentation, CoRR abs/1903.05700 (2019). arXiv:1903.05700.
URL http://arxiv.org/abs/1903.05700

[58] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, G. Vigna, Sok: (state of) the art of war: Offensive techniques in binary analysis, in: SP 2018, 2016.

[59] L. de Moura, N. Bjørner, Z3: An efficient smt solver, in: TACAS 2008, 2008.

[60] B. Yadegari, S. Debray, Symbolic execution of obfuscated code, in: CCS '15, 2015.

[61] K. Sen, D. Marinov, G. Agha, Cute: A concolic unit testing engine for c, SIGSOFT Softw. Eng. Notes 30 (5) (2005) 263–272.

[62] L. de Moura, G. O. Passmore, The Strategy Challenge in SMT Solving, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 15–44.

[63] C. M. Wintersteiger, Y. Hamadi, L. Moura, Efficiently solving quantified bit-vector formulas, Form. Methods Syst. Des. 42 (1) (2013) 3–23.

[64] N. Gálvez Ramírez, Y. Hamadi, E. Monfroy, F. Saubion, Towards automated strategies in satisfiability modulo theory, in: EuroGP 2018, 2016.

[65] X. Yan, J. Han, gspan: graph-based substructure pattern mining, in: ICDM 2002, 2002.

[66] P. Saxena, P. Poosankam, S. McCamant, D. Song, Loop-extended symbolic execution on binary programs, in: ISSTA '09, 2009.

[67] P. Boonstoppel, C. Cadar, D. Engler, Rwset: Attacking path explosion in constraint-based test generation, in: TACAS 2008, 2008.

[68] D. Xu, J. Ming, D. Wu, Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping, in: SP 2017, 2017.

[69] J. Ming, D. Wu, G. Xiao, J. Wang, P. Liu, Taintpipe: Pipelined symbolic taint analysis, in: 24th USENIX Security Symposium (USENIX Security 15), USENIX Association, Washington, D.C., 2015, pp. 65–80.

[70] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, G. Vigna, Driller: Augmenting fuzzing through selective symbolic execution, in: 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016, 2016.

[71] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, X. Wang, Effective and efficient malware detection at the end host, in: Proceedings of the 18th Conference on USENIX Security Symposium, SSYM'09, USENIX Association, Berkeley, CA, USA, 2009, pp. 351–366.
URL http://dl.acm.org/citation.cfm?id=1855768.1855790

[72] X. Ugarte-Pedrero, M. Graziano, D. Balzarotti, A close look at a daily dataset of malware samples, ACM Trans. Priv. Secur. 22 (1) (2019) 6:1–6:30.

[73] I. Gashi, B. Sobesto, S. Mason, V. Stankovic, M. Cukier, A study of the relationship between antivirus regressions and label changes, in: ISSRE 2013, 2013, pp. 441–450.

[74] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 14 (2) (2008) 131.

[75] K. S. Cristian Cadar, Symbolic execution for software testing: Three decades later, Communications of the Association for Computing Machinery (CACM 2013) 56 (2) (2013) 82–90.

[76] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, I. Finocchi, A survey of symbolic execution techniques, CoRR abs/1610.00502 (2016). arXiv:1610.00502.
URL http://arxiv.org/abs/1610.00502

[77] Y. Li, Z. Su, L. Wang, X. Li, Steering symbolic execution to less traveled paths, in: ACM SigPlan Notices, Vol. 48, ACM, 2013, pp. 19–32.

[78] I. Erete, A. Orso, Optimizing constraint solving to better support symbolic execution, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, IEEE, 2011, pp. 310–315.

[79] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, M. Chechik, Symbolic optimization with smt solvers, SIGPLAN Not. 49 (1) (2014) 607–618. doi:10.1145/2578855.2535857.
URL http://doi.acm.org/10.1145/2578855.2535857

[80] K. Sen, G. Necula, L. Gong, W. Choi, Multise: Multi-path symbolic execution using value summaries, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 842–853.

[81] B. Sun, Q. Li, Y. Guo, Q. Wen, X. Lin, W. Liu, Malware family classification method based on static feature extraction, in: ICCC 2017, 2017.

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

[83] R. Tian, R. Islam, L. Batten, S. Versteeg, Differentiating malware from cleanware using behavioural analysis, in: MALWARE 2010, 2010.

[84] R. Islam, R. Tian, L. M. Batten, S. Versteeg, Classification of malware based on integrated static and dynamic features, Journal of Network and Computer Applications 36 (2) (2013) 646 – 656.

[85] Y. Ding, X. Xia, S. Chen, Y. Li, A malware detection method based on family behavior graph, Computers & Security 73 (2018) 73–86. doi:10.1016/j.cose.2017.10.007.
URL https://doi.org/10.1016/j.cose.2017.10.007

[86] Y. Park, D. Reeves, V. Mulukutla, B. Sundaravel, Fast malware classification by automated behavioral graph matching, in: Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research, CSIIRW '10, ACM, New York, NY, USA, 2010, pp. 45:1–45:4. doi:10.1145/1852666.1852716.
URL http://doi.acm.org/10.1145/1852666.1852716

[87] K. Rieck, T. Holz, C. Willems, P. Düssel, P. Laskov, Learning and classification of malware behavior, in: D. Zamboni (Ed.), Detection of Intrusions and Malware, and Vulnerability Assessment, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 108–125.

[88] J. Kinable, O. Kostakis, Malware classification based on call graph clustering, Journal in Computer Virology 7 (4) (2011) 233–245. doi:10.1007/s11416-011-0151-y.
URL https://doi.org/10.1007/s11416-011-0151-y

[89] F. Toffalini, I. Homoliak, A. Harilal, A. Binder, M. Ochoa, Detection of masqueraders based on graph partitioning of file system access events, in: 2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018, 2018, pp. 217–227. doi:10.1109/SPW.2018.00037.
URL https://doi.org/10.1109/SPW.2018.00037

[90] L. Xu, D. Zhang, M. A. Alvarez, J. A. Morales, X. Ma, J. Cavazos, Dynamic android malware classification using graph-based representations, in: 2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud), 2016, pp. 220–231. doi:10.1109/CSCloud.2016.27.

[91] H. Zhou, W. Zhang, F. Wei, Y. Chen, Analysis of android malware family characteristic based on isomorphism of sensitive api call graph, in: 2017 IEEE Second International Conference on Data Science in Cyberspace (DSC), 2017, pp. 319–327. doi:10.1109/DSC.2017.77.

[92] T. Gao, W. Peng, D. Sisodia, T. K. Saha, F. Li, M. Al Hasan, Android malware detection via graphlet sampling, IEEE Transactions on Mobile Computing 18 (12) (2019) 2754–2767. doi:10.1109/TMC.2018.2880731.

[93] X. Ge, Y. Pan, Y. Fan, C. Fang, Amdroid: Android malware detection using function call graphs, in: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C), 2019, pp. 71–77. doi:10.1109/QRS-C.2019.00027.

[94] N. Huang, M. Xu, N. Zheng, T. Qiao, K. R. Choo, Deep android malware classification with api-based feature graph, in: 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), 2019, pp. 296–303. doi:10.1109/TrustCom/BigDataSE.2019.00047.

[95] W. Luo, O. Wei, WAP: sat-based computation of minimal cut sets, in: 28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017, 2017, pp. 146–151. doi:10.1109/ISSRE.2017.13.
URL https://doi.org/10.1109/ISSRE.2017.13

[96] J. Lee, M. Cho, K. M. Lee, A graph matching algorithm using data-driven markov chain monte carlo sampling, in: 2010 20th International Conference on Pattern Recognition, 2010, pp. 2816–2819. doi:10.1109/ICPR.2010.690.

[97] M. Zaslavskiy, F. Bach, J. Vert, A path following algorithm for the graph matching problem, IEEE Transactions on Pattern Analysis and Machine Intelligence 31 (12) (2009) 2227–2242. doi:10.1109/TPAMI.2008.245.

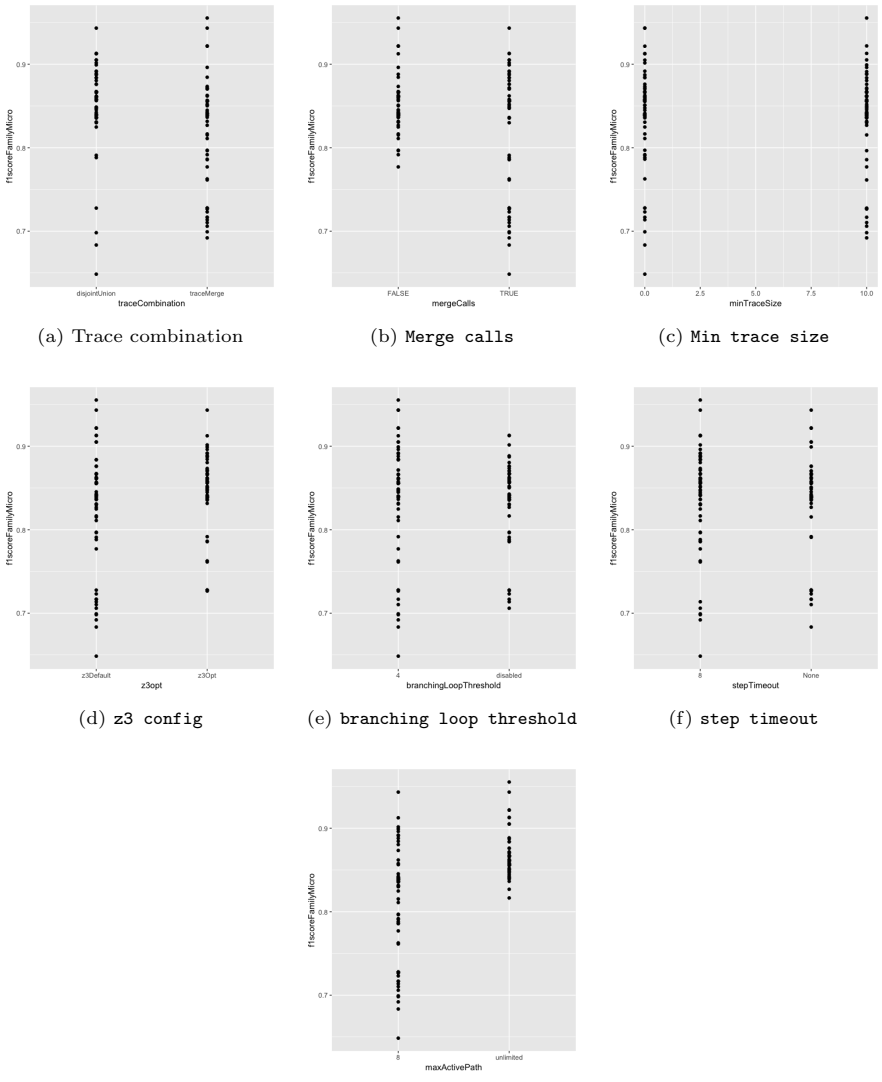
[98] A. Oliveira, R. J. Sassi, Behavioral Malware Detection Using Deep Graph Convolutional Neural Networks (11 2019). doi:10.36227/techrxiv.10043099.v1.
URL https://www.techrxiv.org/articles/Behavioral_Malware_Detection_Using_Deep_Graph_Convolutional_Neural_Networks/10043099

Appendix A.

Appendix A.1. Main Effect of Graph and Execution Heuristics on the F-score

In Figure A.6, the impact on the F-score of the seven studied factors (i.e., the graph building and execution heuristics) is reported. The plots aim at representing how classifier and the levels of a given factor are related.

In each plot, points on a column represent the micro average F-score achieved by all the configurations having a specific setting (on the x-axis) for the heuristic reported in the caption.



(a) Trace combination (b) `Merge calls` (c) `Min trace size`

(d) `z3 config` (e) `branching loop threshold` (f) `step timeout`

(g) `max active path`

Figure A.6: Main effect of the factors. Each plot represents the levels for a given factor on the x-axis. On the y-axis the quality of the classifier (F_1 score by family in our case)

Appendix A.2. Interactions Between Factors

The interactions between each possible combination of factors are represented by means of the Pareto plot in Figure A.7. Basically, such a plot allows one to observe factors and interaction effects having the highest impact on the performance (the F-score in our case). The impact, positive or negative (respectively in black and gray in Figure A.7), of each combination of factors on the F-score is on the x-axis.

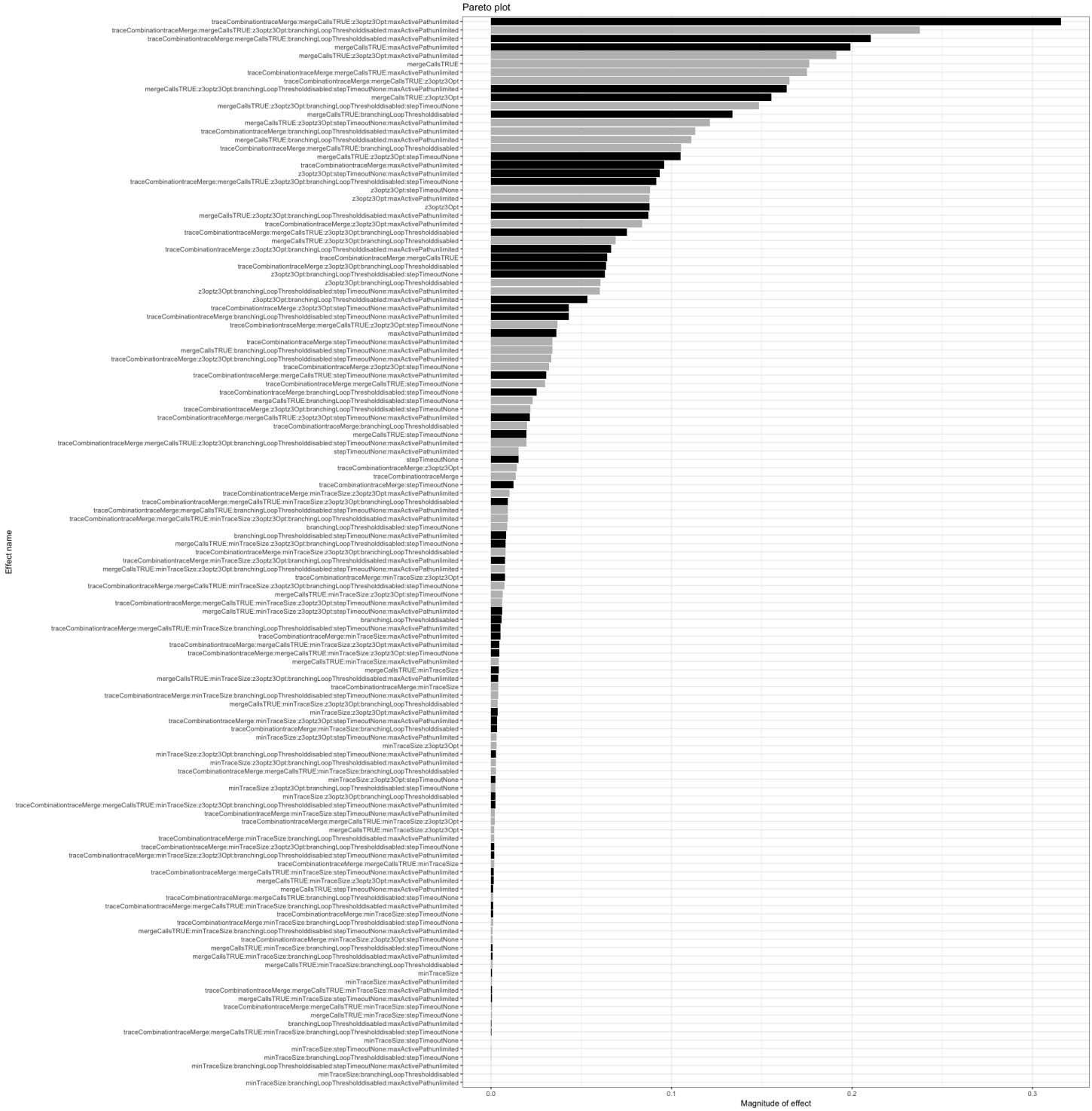
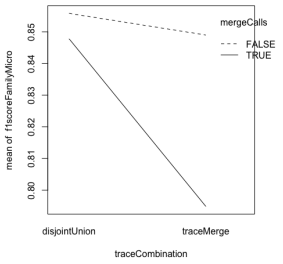
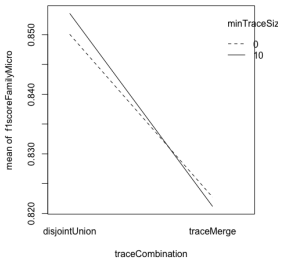


Figure A.7: Pareto plot for the factorial experiments. In black positive coefficients, in gray the negative ones.

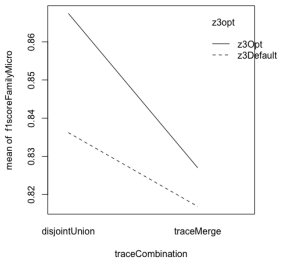
Interaction plots in Figures A.8, A.9 and A.10 provide a more detailed view by showing the relations between each pair of factors (by means of a linear model).



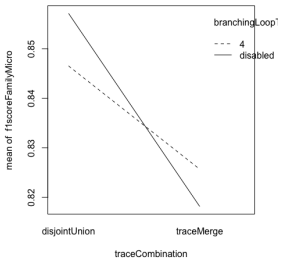
(a) Trace combination and merge calls



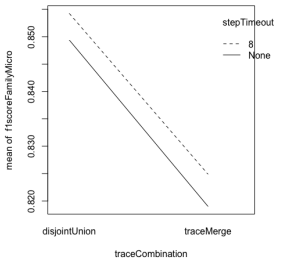
(b) Trace combination and min-trace-size



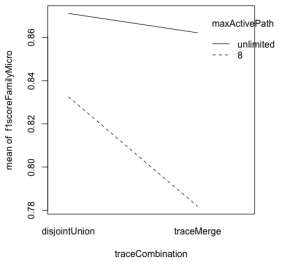
(c) Trace combination and z3 configuration



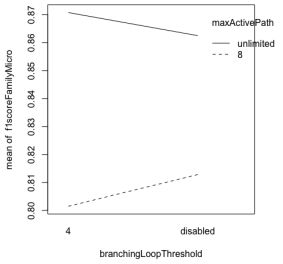
(d) Trace combination and branching loop



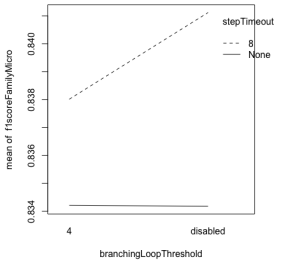
(e) Trace combination and step timeout



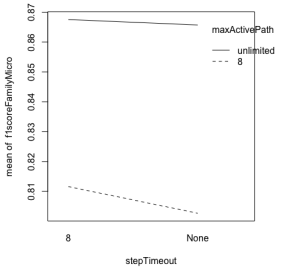
(f) Trace combination and max active paths



(g) Branching loop and max active paths

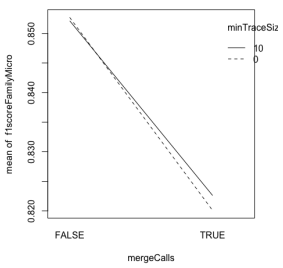


(h) Branching loop and step timeout

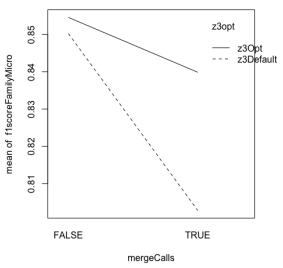


(i) Step timeout and max active path

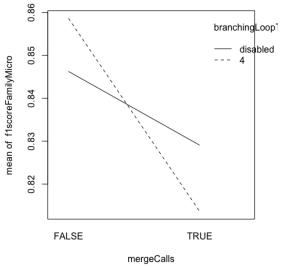
Figure A.8: Interaction plots for the ANOVA analysis (1/3): interaction effects on the mean F_1 score by malware family for each possible pair of factors



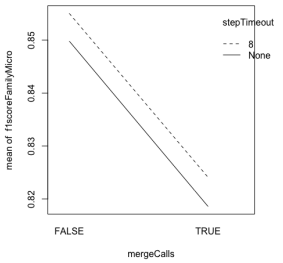
(a) Merge calls and min trace size



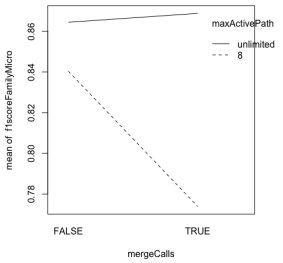
(b) Merge calls and z3 configuration



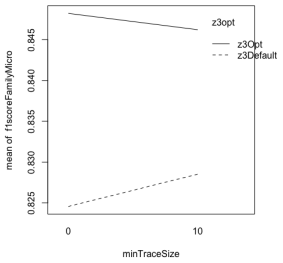
(c) Merge calls and branching loop



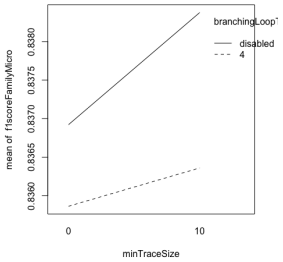
(d) Merge calls and step timeout



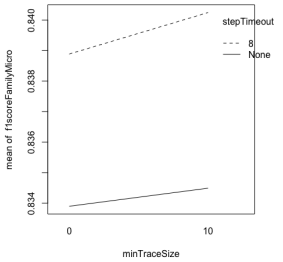
(e) Merge calls and max active paths



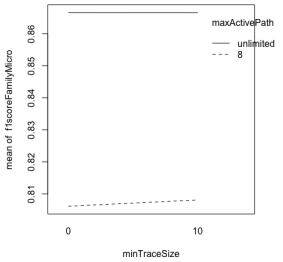
(f) Min trace size and z3 configuration



(g) Min trace size and branching loop

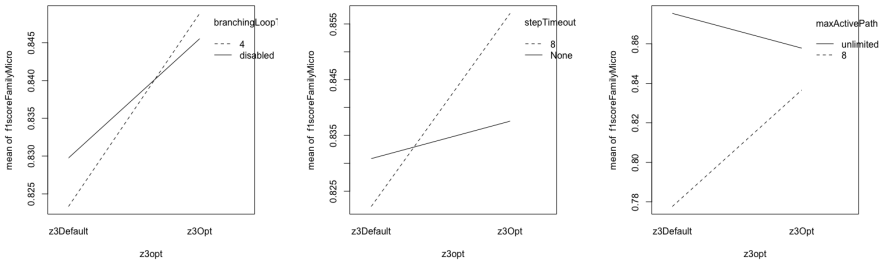


(h) Min trace size and step timeout



(i) Min trace size and max active path

Figure A.9: Interaction plots for the ANOVA analysis (2/3): interaction effects on the mean F_1 score by malware family for each possible pair of factors



(a) z3 configuration and branching loop

(b) z3 configuration and step timeout

(c) z3 configuration and max active paths

Figure A.10: Interaction plots for the ANOVA analysis (3/3): interaction effects on the mean F_1 score by malware family for each possible pair of factors

Appendix A.3. Relationship Between Connected Components and F-score

During our analysis we pointed out that the litmus test for the quality of an SCDG-based classifier is the presence and the size of connected components (see Section 5.2). Here we report, by means of box plots the relationship between the performance of the classifier (with respect to the binary and multi-class classifiers) and number (see Figures A.11, A.12, and A.13) and size (see Figures A.14, A.15, and A.16) of the largest connected component in the set of SCDG-based signatures used by the classifier, for all the 128 experimental units analyzed in this work.

In the plots, on the secondary y-axis the red stars refer to the F-score of the multi-class classification (malware classifier), whereas the triangles the one of the binary classification (malware detection). For the sake of readability, the 128 configurations have been grouped in sets of 32 for each plot.

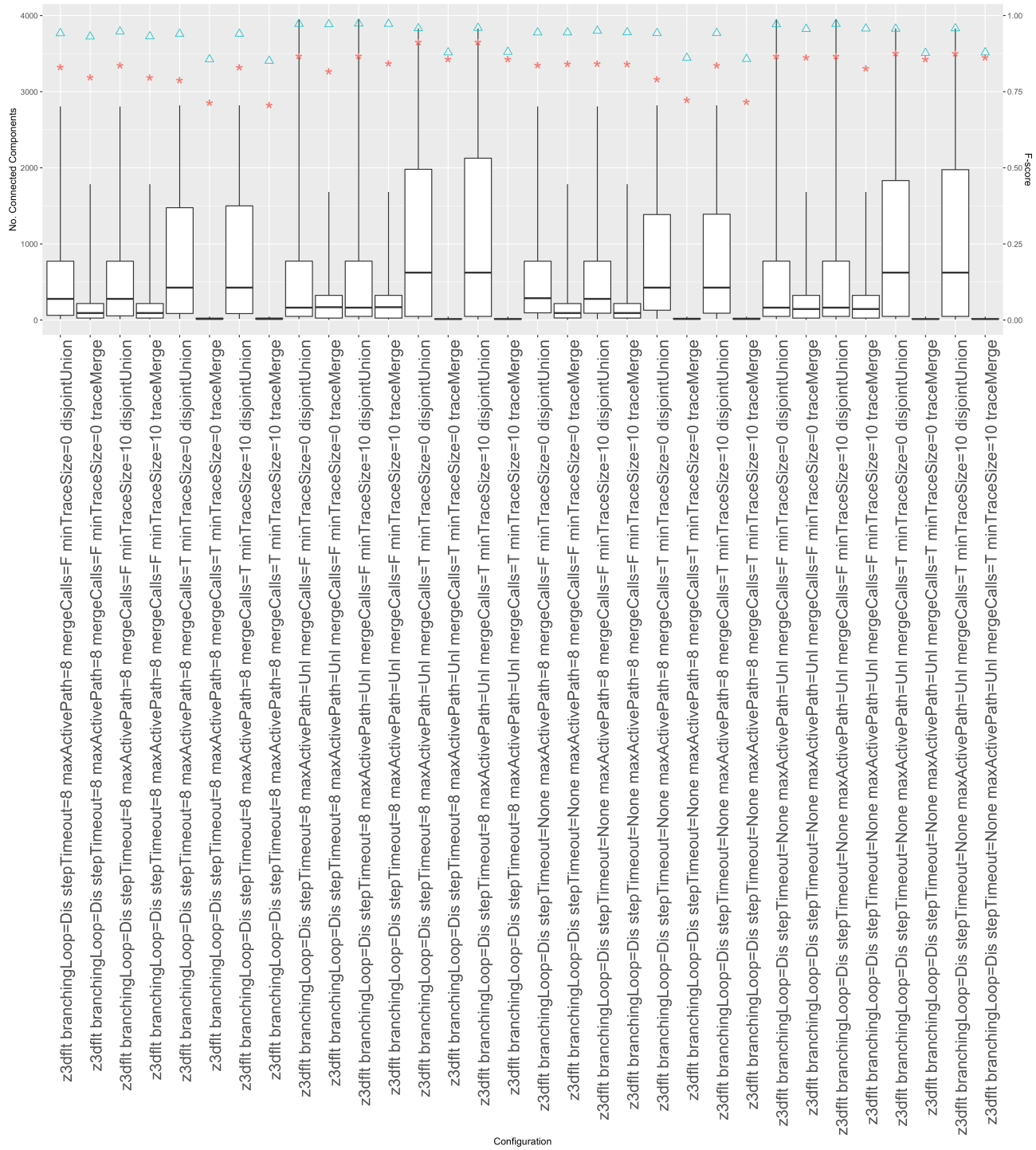


Figure A.11: Number of connected components for the SCDGs in each configuration and F-scores (triangles represent the binary classifier, stars the multi-class classifier) - (1/3).

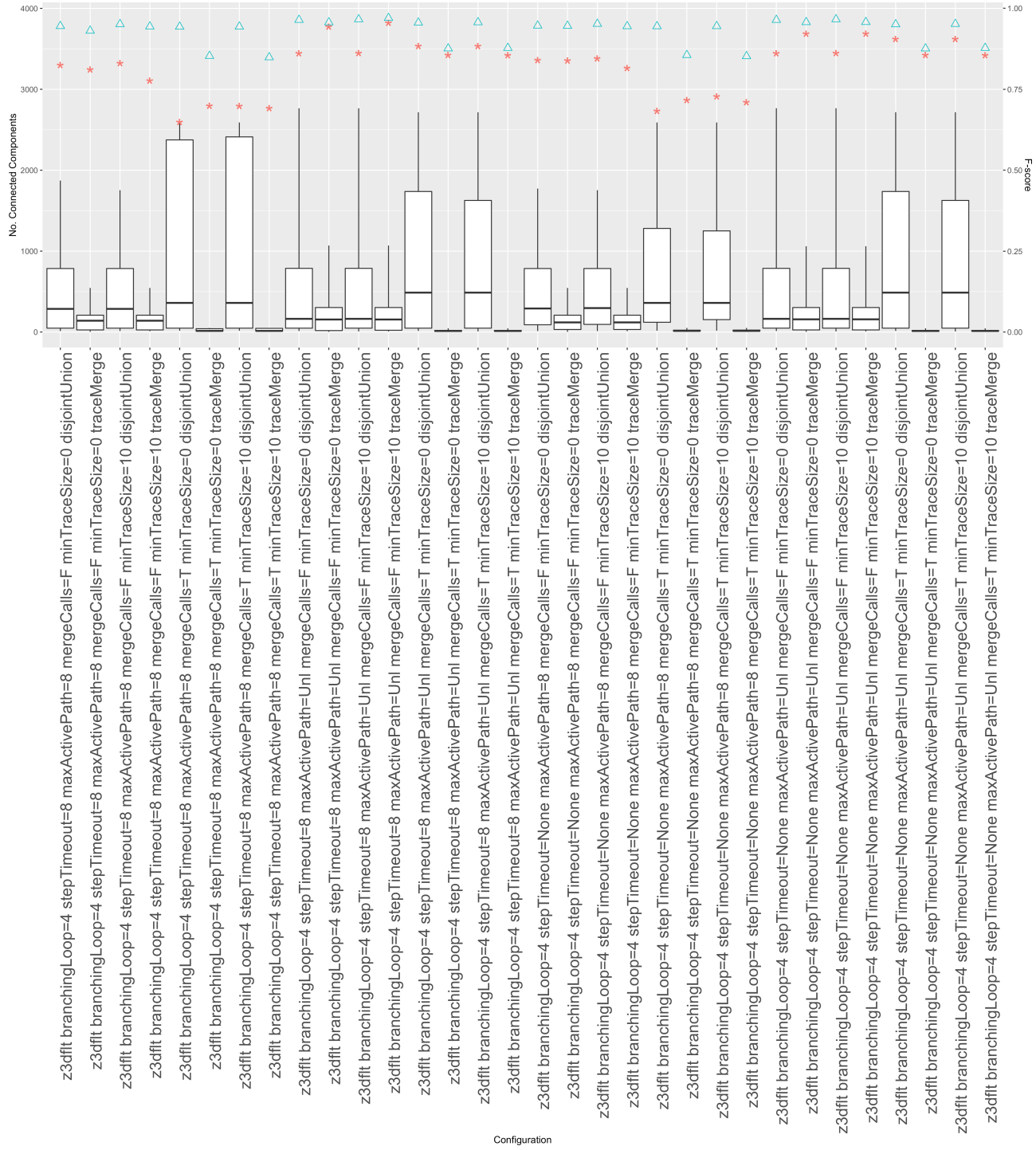


Figure A.12: Number of connected components for the SCDGs in each configuration and F-scores (triangles represent the binary classifier, stars the multi-class classifier) - (2/3).

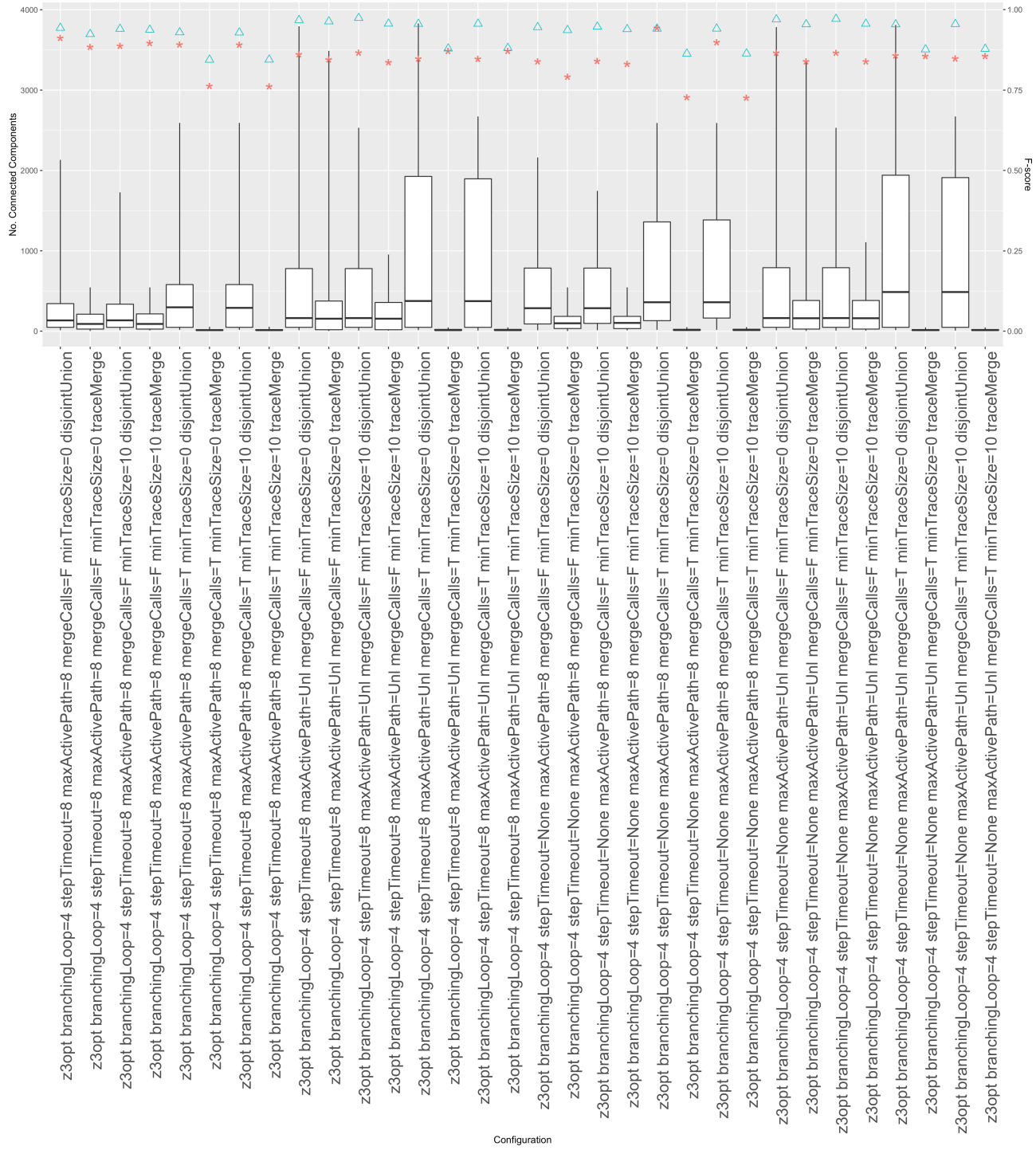


Figure A.13: Number of connected components for the SCDGs in each configuration and F-scores (triangles represent the binary classifier, stars the multi-class classifier) - (3/3).

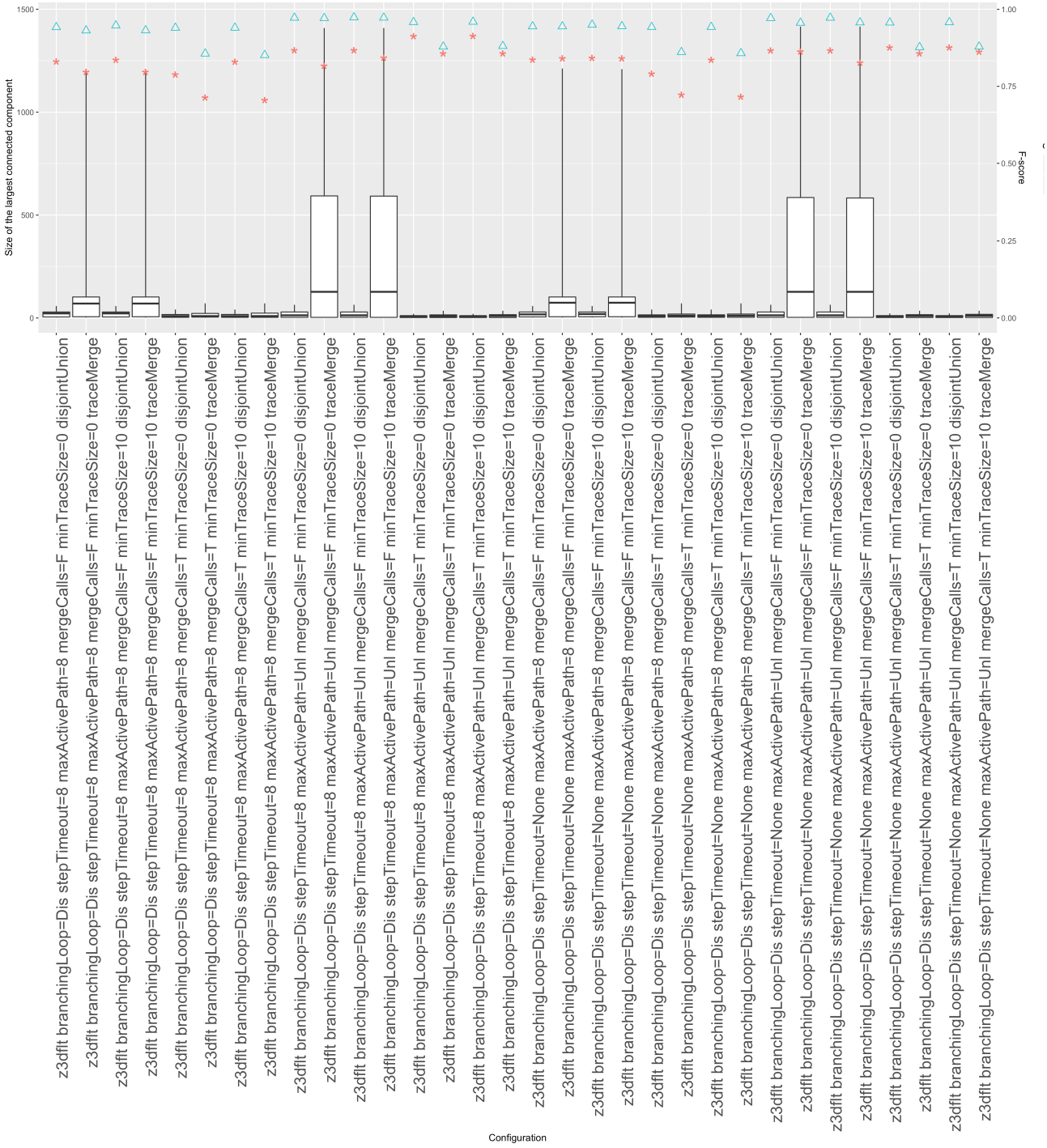


Figure A.14: Size of the largest connected component for the SCDGs in each configuration and F-scores (triangles represent the binary classifier, stars the multi-class classifier) - (1/3).

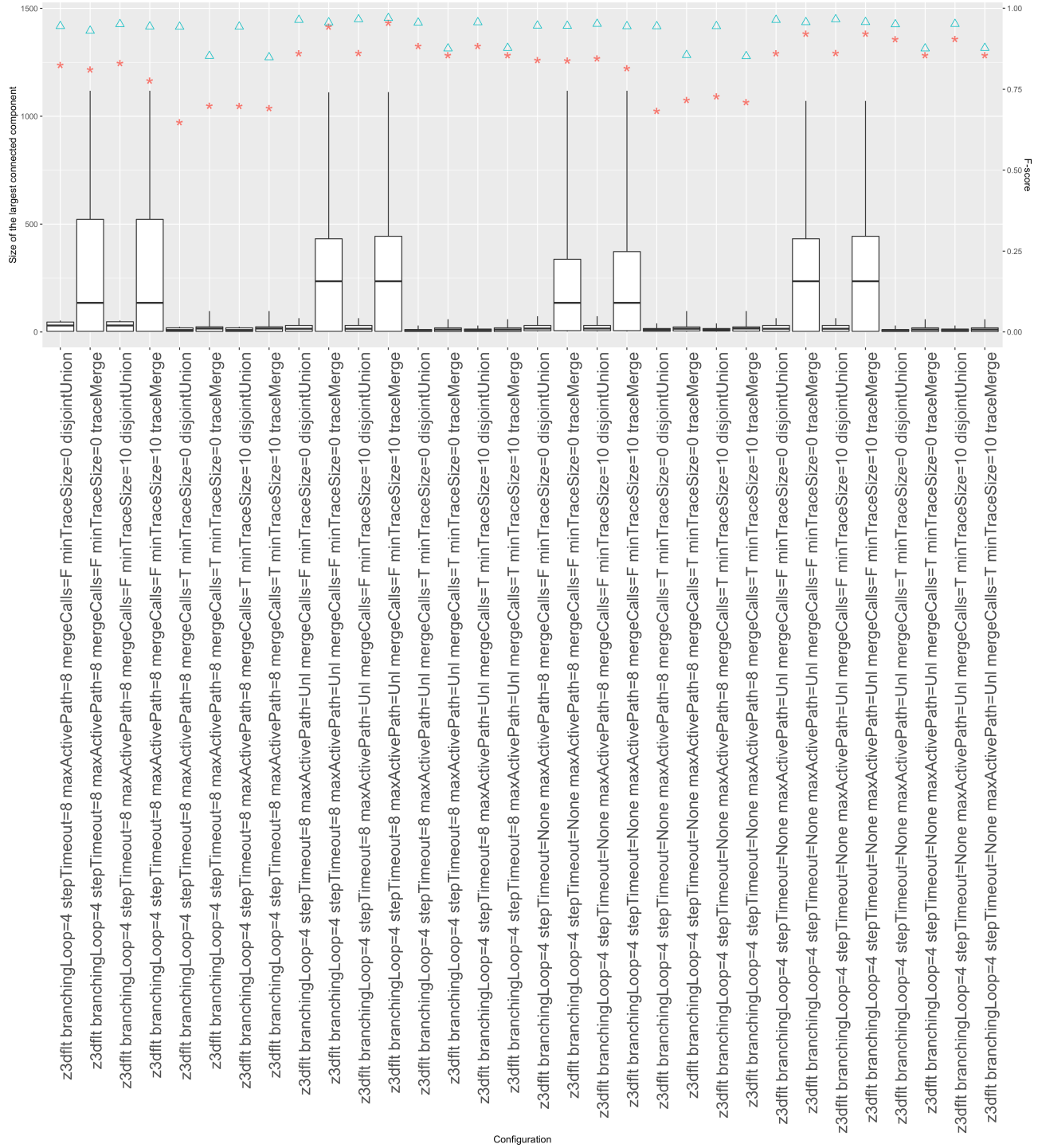


Figure A.15: Size of the largest connected component for the SCDGs in each configuration and F-scores (triangles represent the binary classifier, stars the multi-class classifier) - (2/3).

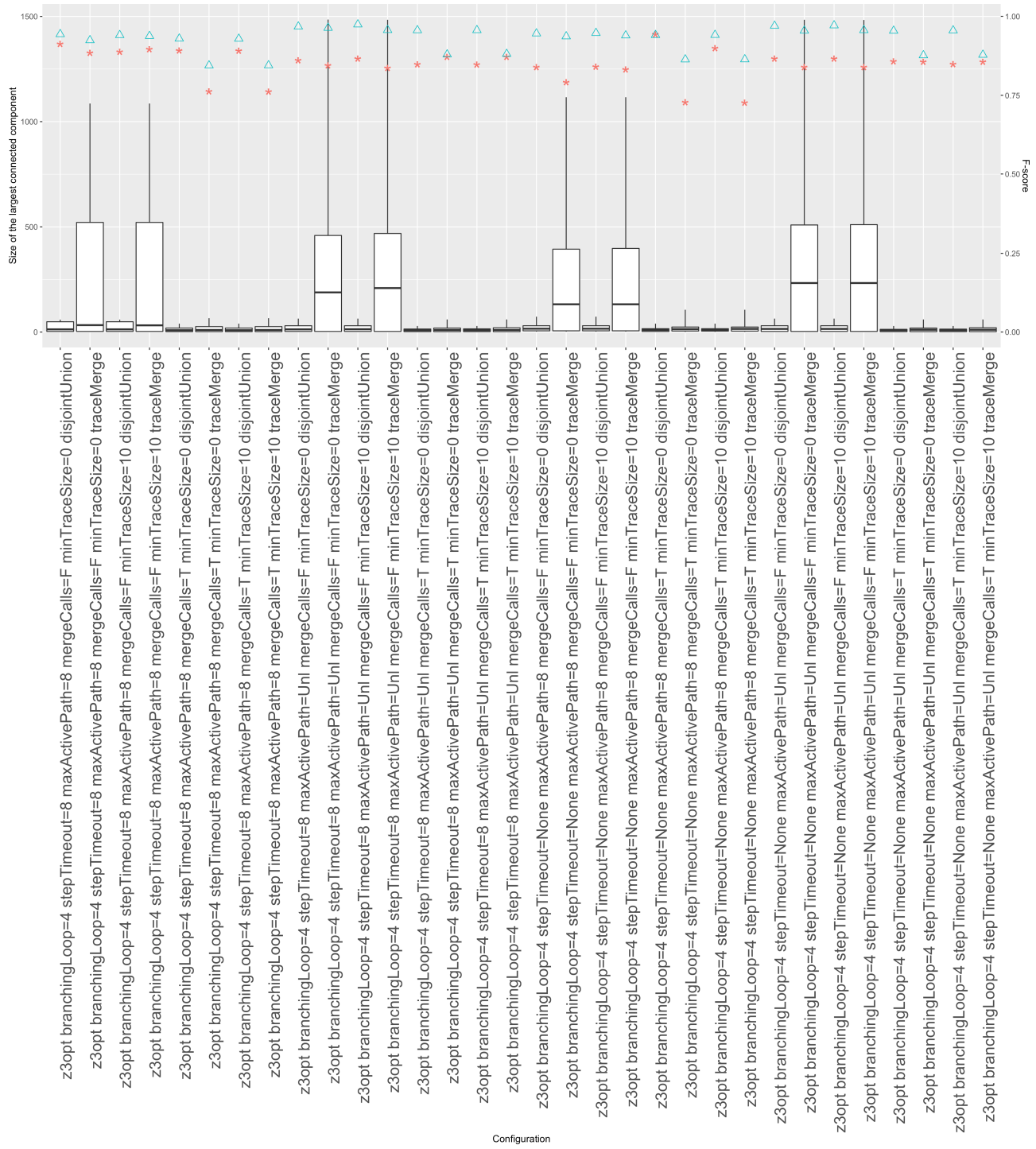
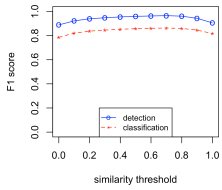


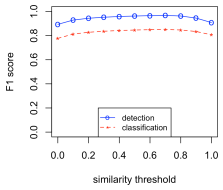
Figure A.16: Size of the largest connected component for the SCDGs in each configuration and F-scores (triangles represent the binary classifier, stars the multi-class classifier) - (3/3).

Appendix A.4. Impact of the Graph Similarity Threshold on the F-score

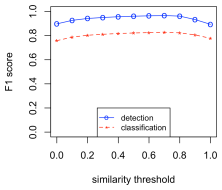
In this work, according to our previous experience and after the preliminary analysis, we have chosen to set the similarity threshold for the graph matching algorithm gSpan to 0.7. In the following (see Figures A.17, A.18, A.19, A.20, A.21, A.22, A.23, and A.24), we report the F-score for the binary (malware detection) and multi-class (malware classification) classifiers, for all the 128 studied configurations, by varying the similarity threshold with steps of 0.1 in the range from 0 to 1. Here the blue circles indicate binary classification, and the red stars multi-class classification.



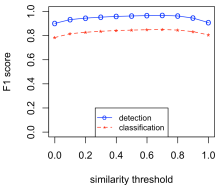
(a) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge



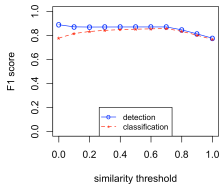
(b) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge



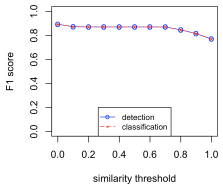
(c) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge



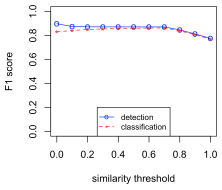
(d) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge



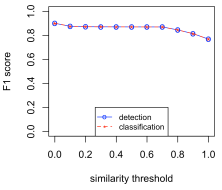
(e) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge



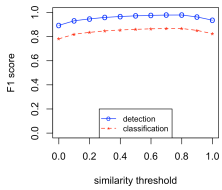
(f) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge



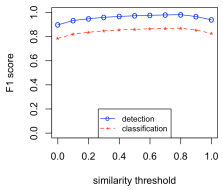
(g) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge



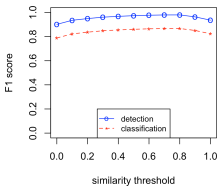
(h) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge



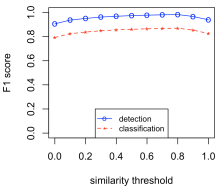
(i) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion



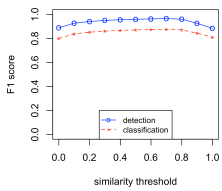
(j) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion



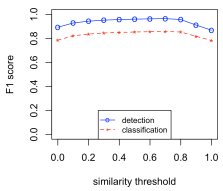
(k) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion



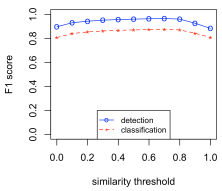
(l) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion



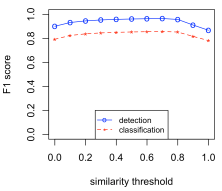
(m) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion



(n) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion

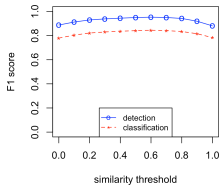


(o) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

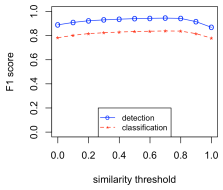


(p) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

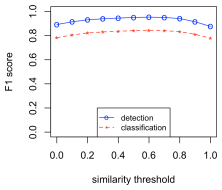
Figure A.17: Graph similarity vs. F-scores (1/8)



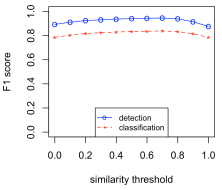
(a) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge



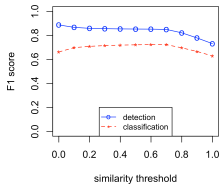
(b) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge



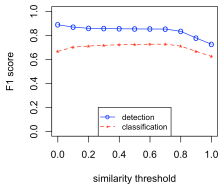
(c) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge



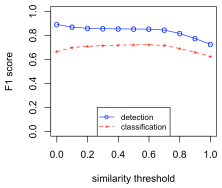
(d) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge



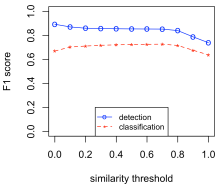
(e) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge



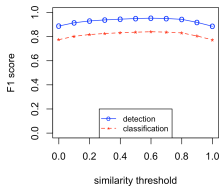
(f) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge



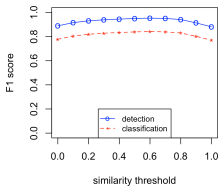
(g) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge



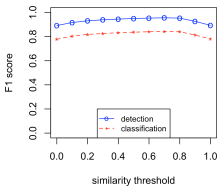
(h) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge



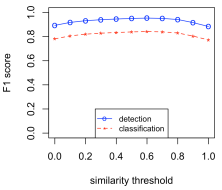
(i) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion



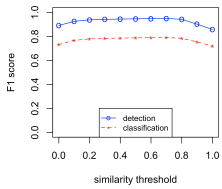
(j) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion



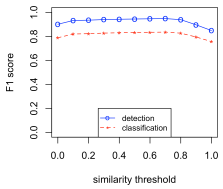
(k) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion



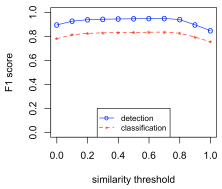
(l) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion



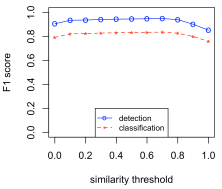
(m) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion



(n) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion

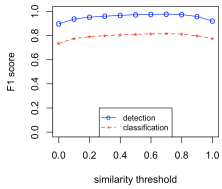


(o) z3dflt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion

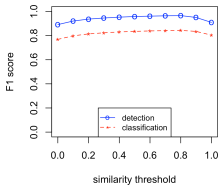


(p) z3opt, branchingLoop=Dis, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion

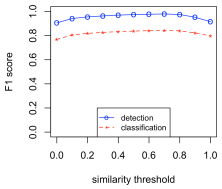
Figure A.18: Graph similarity vs. F-scores (2/8)



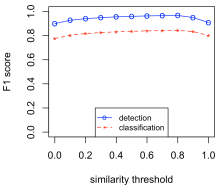
(a) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge



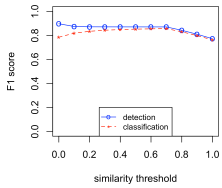
(b) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge



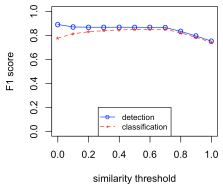
(c) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge



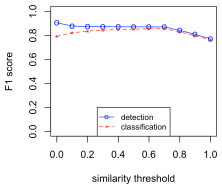
(d) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge



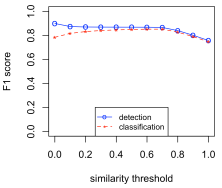
(e) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge



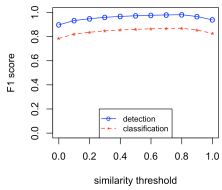
(f) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge



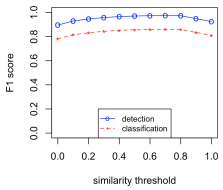
(g) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge



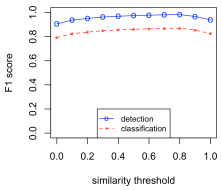
(h) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge



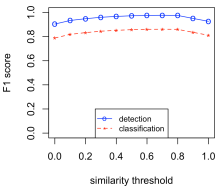
(i) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion



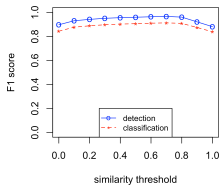
(j) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion



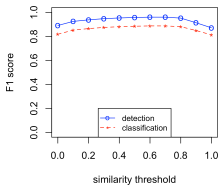
(k) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion



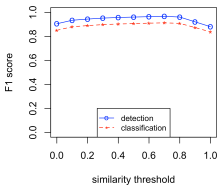
(l) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion



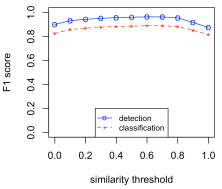
(m) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion



(n) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion

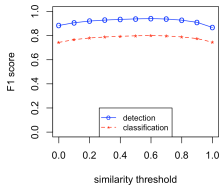


(o) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

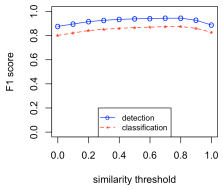


(p) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

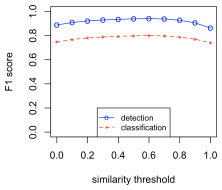
Figure A.19: Graph similarity vs. F-scores (3/8)



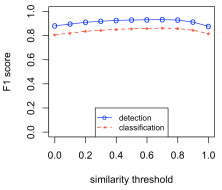
(a) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge



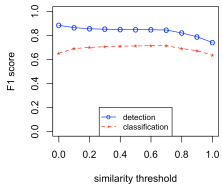
(b) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge



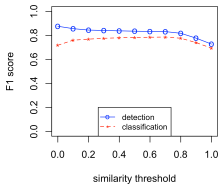
(c) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge



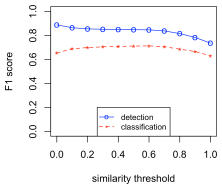
(d) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge



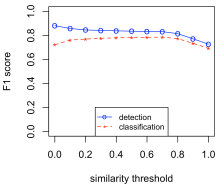
(e) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge



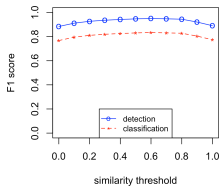
(f) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge



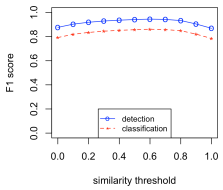
(g) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge



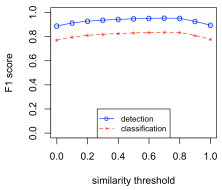
(h) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge



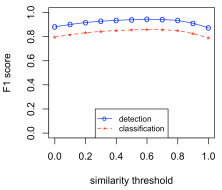
(i) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion



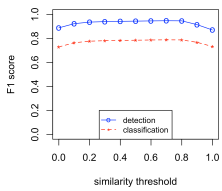
(j) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion



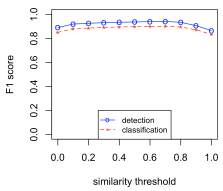
(k) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion



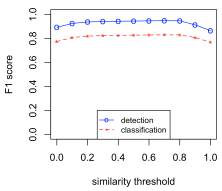
(l) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion



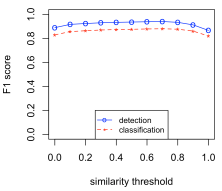
(m) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion



(n) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion

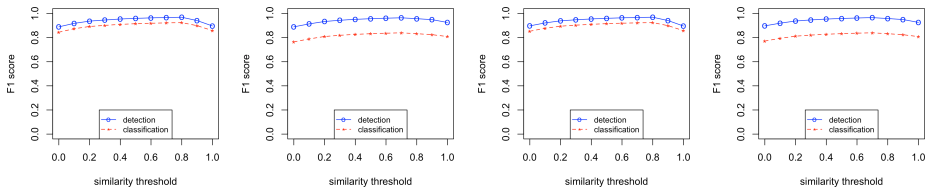


(o) z3dflt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion



(p) z3opt, branchingLoop=Dis, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion

Figure A.20: Graph similarity vs. F-scores (4/8)

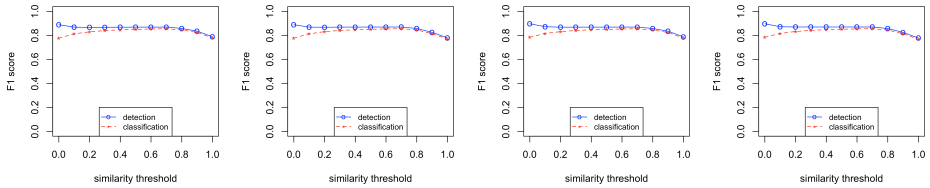


(a) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge

(b) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge

(c) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge

(d) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge

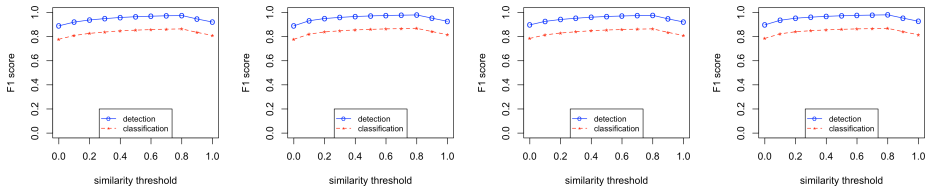


(e) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge

(f) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge

(g) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge

(h) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge

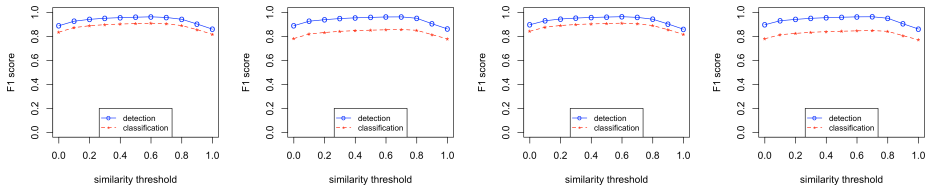


(i) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion

(j) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion

(k) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion

(l) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion



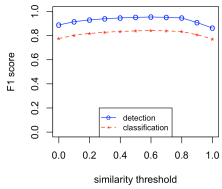
(m) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion

(n) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion

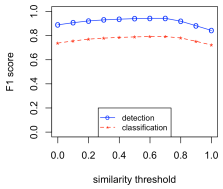
(o) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

(p) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

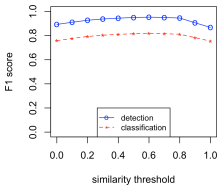
Figure A.21: Graph similarity vs. F-scores (5/8)



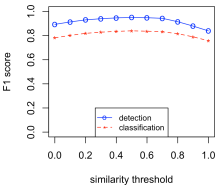
(a) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge



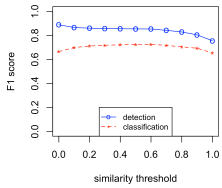
(b) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge



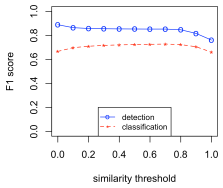
(c) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge



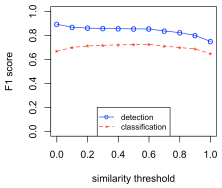
(d) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge



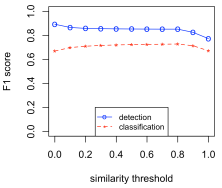
(e) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge



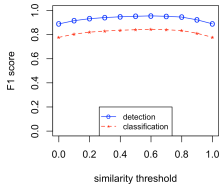
(f) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge



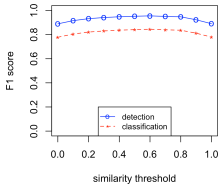
(g) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge



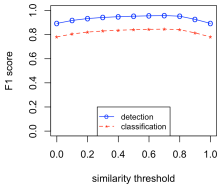
(h) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge



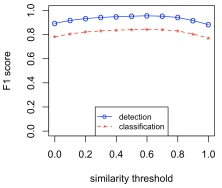
(i) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion



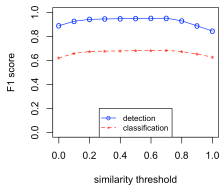
(j) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion



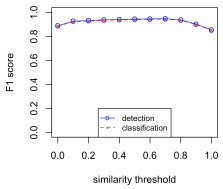
(k) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion



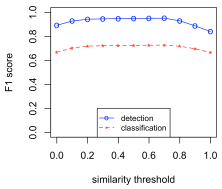
(l) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion



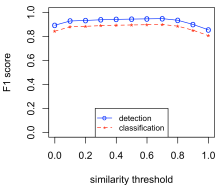
(m) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion



(n) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion

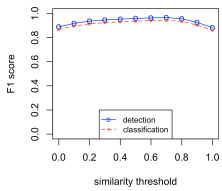


(o) z3dflt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion

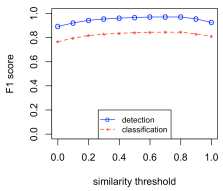


(p) z3opt, branchingLoop=4, stepTimeout=None, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion

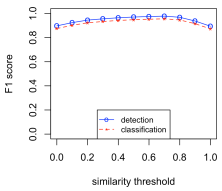
Figure A.22: Graph similarity vs. F-scores (6/8)



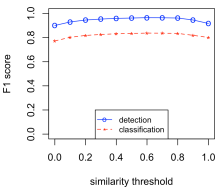
(a) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge



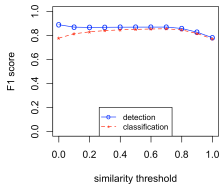
(b) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, traceMerge



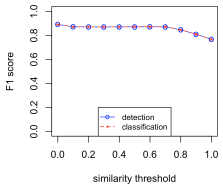
(c) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge



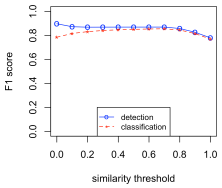
(d) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, traceMerge



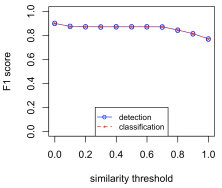
(e) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge



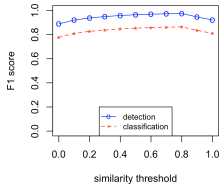
(f) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, traceMerge



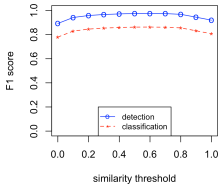
(g) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge



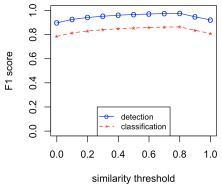
(h) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, traceMerge



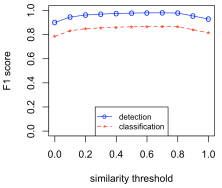
(i) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion



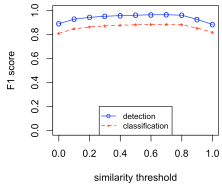
(j) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=0, disjointUnion



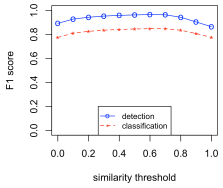
(k) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion



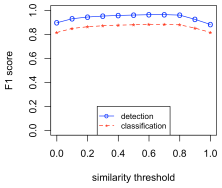
(l) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=F, minTraceSize=10, disjointUnion



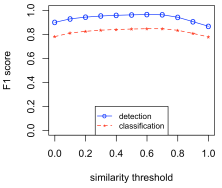
(m) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion



(n) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=0, disjointUnion

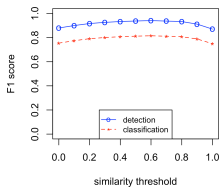


(o) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

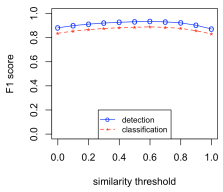


(p) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=Unl, mergeCall=T, minTraceSize=10, disjointUnion

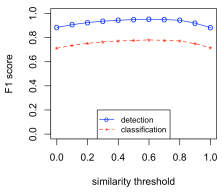
Figure A.23: Graph similarity vs. F-scores (7/8)



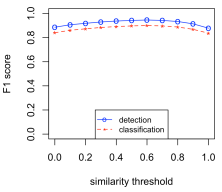
(a) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge



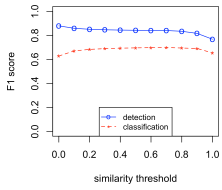
(b) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, traceMerge



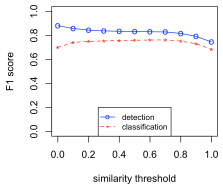
(c) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge



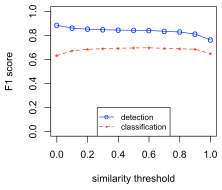
(d) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, traceMerge



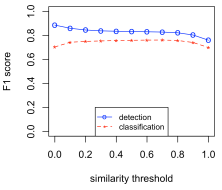
(e) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge



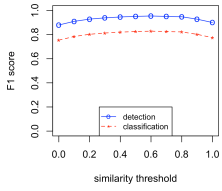
(f) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, traceMerge



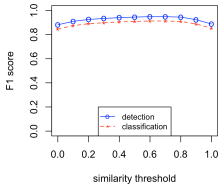
(g) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge



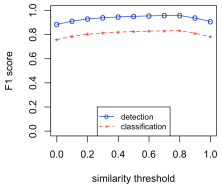
(h) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, traceMerge



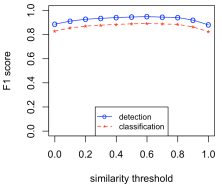
(i) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion



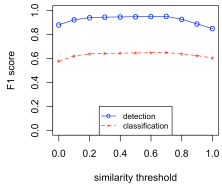
(j) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=0, disjointUnion



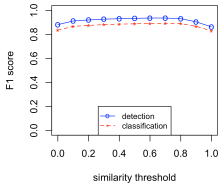
(k) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion



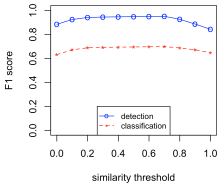
(l) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=F, minTraceSize=10, disjointUnion



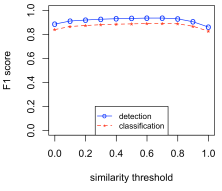
(m) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion



(n) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=0, disjointUnion



(o) z3dflt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion



(p) z3opt, branchingLoop=4, stepTimeout=8, maxActivePath=8, mergeCall=T, minTraceSize=10, disjointUnion

Figure A.24: Graph similarity vs. F-scores (8/8)

Appendix A.5. Dataset

Due to a NDA (Non-Disclosure Agreement) and security concerns we are not allowed to share our dataset (even upon request). Notwithstanding, for the sake of reproducibility, we provide below the SHA1 and the corresponding family for each binary analyzed in our work. Being the SHA1 hash of a file unique, interested researchers can lookup for the samples in other available datasets e.g., on VirusTotal.

Family	SHA1
jeefo	f1f06713bfce019a1d33185bf65dbd3ff4854b34
	12c35183ab2c960234e8b3224fea8653d14d882e
	f59b3f97f596daba094e974b90c37fd111add32a
	7324e63211332c945c0db095317409dba5b8aa89
	4971eae31449da08f120d6d4cab2ddf57c1a5115
	8eb11682498a8291aba5c0836c7992a6dcbe54ed
	667194c6c6f672a0a77d89937073072d3b707862
	31448f16a4845ebfa4864dabc000b54871e03acf
	41e97ad145fb417cd9a214b8e169ea1bb43d5850
	8c5903f36018d0613f70704d0b12adeb9daa4ce2
	6b998c847c35691fdf71adcfed5cc759ecfdd11b
	02c0362de786d276eb5dd67065eef7e49ad6d4cc
	f5b4782760fe2c064ca9d97f0f39b7fcb03957a5
	ae51bc9ed3f9d90ef2bda01fd76d22c97e46cdb6
	bf2d30db511ee0d469de55386bf6e7849439a45c
	ef827d0aefff1572e1816c907cea4b4612142375
	9a04faa46701d9ef53852dd896ec9312b580d9d6
	dce188f3809291ac046a72bb78265b182b2e7c75
	fb030359343a43d4d6ff142589dd58b1db9a8fc2
	1a60a66557f0c4e82629b88ddb18886f6bbeccba
	150b079cdefa6d664a63fe64107b31af1a258350
	c1af341a10eac11cafa0c51a3c7165fb6020eacd
	37f0a683ebfad0af8c590345372c0bc95ec63de5
	0fb629bfdfccaa12405e141a418edc2f13c771ac
	17b547234616c69682cc7c11fa6f8b75626ebfdf
gamemodding	e60663028faa214e2ab493c44019365f6ac7a349
	5bd39ababe69d4023089e53f3ca23a2e55ea1c06
	6f3a5a4f7884363947efe33f66d9ebcfc361e472
	83849d556715a99ccd98b78ec9aa1db783a60251
	641fe0e576267a1a582409283776645104a9270e
	45b1c294e3674883257d4ebdcd75a19f8bb2d751
	6fb04001558d7b9c7e27204c6a338d343f05cc15
	9fac4eed08c9d16e05605116a22c58b47fa6a67f
	44547232398772218535157ff88eea48fed5d7d1
	a003ffe778c404eed9b5f3c8e721fe1afc81a40f
	d3f6956727f02b5b832aecf4afc6792c873aa118
	73f0b47703243aa61d3ed3282e62a90bb68df102
	66a422dffe441a3a3f3347044078cc5c70d05353
	5f9f64952b42f1b95fb47fbe99a4dd3e511712b8
	46e9671f16a363f2033d1756ce171032107b5a48
	614dcd5d165edae6d9cdba83303f1984c74addb3
	7e528bb6a4c608bff1175061aaeae84061008054
	f6a41df6b8442e9c5b2a9b5bb3821872b6e3a25e
	2bb11518070e7d7e4ba1b02bb56cd4140eae3c5a
	fa7824cf823126910eb1c8e06276bbb31d117c12
	fb4533c5aeb3df9a4d0278a7d3fb588da4cb71c3
	41b6b8e2f24202503328c417e502a02105419684
	6342e8acbafe064d22753400f2696140e91e8773
	0c33ba4fc177e2c2c017ea224b22b60ff48357d4
	875c71ce5f363e6fd90d20d8d8eea838a2c3926e

Table A.4: Evaluation dataset (1/5).

Family	SHA1
mira	09cebe61bd8a85fd1b8e15a87bc528e50e9146d1 8b171470662a7f77c711d1f8219f742b67bf6636 8be30a111087292144c4a105b1e59336c646c7b2 9f93c3603f91455ffa10611ed505682ad3750222 f2678a18b5760e50e1496cc10793b8dd0d43babb 8735fb97613720390ff44ec4088e19a6bf20fc82 7301f70dab808be9cb9d61eb829c0ac84e22fed6 22531cac47269c6b542700e11686ba310b474425 4c2716baa548f10a97cb581ac3b3ff22970dbad3 d1dac653c287606c58c1344a21bc98ff1cab33ae dbb489de8a845ce8cec003f9a73c7b848dc4c04d 69dff1bdad1773ec9aa187b33cd0b1c19e8e3cc9 cab35a276d1631ef5b62a91c0b1fe51e95dd4e7b b74cf183dfdadf124ef29954d855eb0d4493a603 8777e4a3528e399e94223f4f8b0e9d0c29fe6c0a ec3fa64c6a0778e610b59c9639abed71e408984c 59ac1c3502cce4128fe56e953aa59614275eaf1d 03c60cb6dfe950ab8abfc79db64a4485304ed4b4 f19390b920a515b752f5c5827d7e909e32912caf 4c68e704f180f51f60c5864760837802ae42ed4b 7fb49935dc90a6ca38f3e287f6862f335dfe1ca3 a69aa13f110a258e6be1edfa14c9cf6453440cd7 5ef1d37d3ac6e376719af055956a3f7ccb961b05 b993fca2a7e5475de0503ac6fe53ec502ffdda71 042e5606a5469abefcff6f63f76933459baa3cf5
installbrain	42314c2e1ddc0366aef7e6227f23e170fd34e517 312596700c2326168e08d2c339400e8b740317f1 d5e9fc66f70d40bff172470f3e8c76affaeabba8 2ae5536f48452423d7f32db71460cabd16107df6 a082516a4389503d15f4069922e1b18e2537f750 87a116c78c4cca00024cbbc672ab3a2bb3dec27c 2ce3f5baa7d7f4b2e3a64e8b9e8549c3dfc945a6 dfdb89211d2f15e92cab0cb0f42da063e1d3b840 00364bf2ad09a975efacdd420876d29199a1e388 1e2aeaf7e811fdb550302a0603b0b30fb370f503 5f3272cd501f346e6075a58ba94ff977e241b225 2839fda2772f066150dca6b7d95efa33e85bfc79 667e6f22bf45c38fc33059e9f7399295b2d897f8 8d8e373ed0845ed4ed2d983ee2b1c5b00cad9076 ead0d438833687ba989318c9e27290da5581191b e7be4cb61f06a72cc1b300508608d6689edd68d2 03c437977840383bb01b9581cb3eb343cfa852a4 21029fa0e28350101d038ad6d27ca251666d6b2e c443922da881d8ad5fd3ea0e40ae5ffd4938df6c 9cb055b6f6df6e3e42372b2dc60771f33cf80fb4 26a55c1e24e3429bae6de47e50d7c46cdc4115ce 78339b24ffd2d0e8801b889aa00b8d9255830772 8bf70b22639c9bd822f90778327a521d19e37ca2 7ca87cd015492dea5eeff97d023add97efe277ba fc9d8bd601f53343bf0f5c4116d2f840423eec70

Table A.5: Evaluation dataset (2/5).

Family	SHA1
addrop	ed268e89978f59dbd6a34613a251322b1f7e7f70 9ee17a6fe4da450ab751712264af58bdfe8cdab8 a5216221219a3fa2f0e03f742f62f3c670a68b34 f058a34fa7a0ad2dfcae1edcad9b42bae31e15dd 13dce4594d01c430e3098d5da26bf7cec629ad0c 5c632ad6f5029e05becd4b47924a29c2ba6033af 2ae7f1ef08b8657de3060068bbdcf375e2f20cd3 f8a5215a4bffc2fc90f115e914fb26b5817d50e3 adeda6a02c5312bb12fceaf71a055627f50ba180 9c240d316d0b8766a5cf6ec07e5aa4fac6c819cc d97825ab75f08a1efbc924e2969487a0a6408e7f 65f46c8a82782640e502396a107f4403fbbacc72 8fcd4ffcd87c1283a470aeb5184ae5dafa42fd54 37c443875ebbf6056ccaf22ed6f184e724520336 cfb14ce62e981a610f88be8811e1dc343db12b92 7188044570e204369e3bab83039ee56753a381cb 010b38298a4a4d1e256a27e87cc0d1f34f47b64a 14606a9aada603389684160b59794a766efe9f0b 181433c9aa6780710b44dd1c48ad8796dca18cb8 e0fdd45fdf81ecde6823a4bf4ffccbc35cabd1cb c8ed6e3a1353616d0232b5ce0ec74fdfac61b4b1 3e746871396e56491bcdf24c4d0120e15f98ef37 5d62957020594d5e66d55d4d72921154475ff8a9 c060d2975a5089b6501bbb8839d856ee5dc931a6 e51ecb575bbd772b1e7e7391655e777514d8b8d5
couponmarvel	9e25305bc47f2046e133b2f09b3b828715da811e 07fc7ccac20b4f412876e7a992c340a0a65e2f63 17bb3448fae7c6b3409feaa6002a82348d193949 a5f1500704238efae5084368028111066b89d05d 1ecb611f1c29e9776baf619675fbc44403d534a3 b3ea56b1995e14544202880c0cb211b197a9e95a 96c3b39b61552fbe511ee7ada13a29334b4aa286 c5c19840d6887f61e0e10df92ee95adb3af2ea6d eaa9a1a1383cac81e36189d9324aad0028dc68a1 d2d5fe3d8ed8a4f0572c36e99e86226b0d827d24 d94f7fecac0c27903ded491b233dfbb2a17e647b 6828ec0826075baab6186545748db93677689b41 2aeef97c7984303e3c02a0b70701ed2702dac98c 6e04ed657682a9ba46bb8e1c19e26bd750722655 aa55f46ed48b9be0989a3cbb58191bbd99533cc2 9486e83fd4ca8f75e8c57135fb4e31d4849b65e3 467cd38943f87ffabe4a848efa7da3f2c9bf95e8 3e99db71a9b35285c920d576c989c184d5387e5b 3b08d213f1938280cb49a215875a724eb40a7983 6723086317f9721401ee2d874af7d9c32a831364 8a95de1e8776d2ae66ab3d2d83148e59d3d27b0c 363d98aab25bd882fb7732af3ec21d92cbeb483e afbc4811e111cf1df0d6740aef4f0ce57a0707e3 4840f4f717359e13ff5d600021d037b4b85ee5b6 5d165d2374bf547f299180ff8abcb8e2372b5906

Table A.6: Evaluation dataset (3/5).

Family	SHA1
multiplug	2cac3884cca13c4a24caf24235d5f0006913ea26 63e138c4b9af4a951c753de8866328eec8cbe3d4 a6f9db9d1717013e2dfe8e47e24e25f831ed5278 7e91a2248abed93eafebfb2bf05810bb0b32802e d9a8f8e3d97e6d47045c4209371a424bdd977026 01531ada6cc172dfa4e86d826222b09090cd956a afb72959f2b0b7d5f131130bc5fe21f9bde9c9ec 0479e8499c1a3e94da5231e7aa47eff0d84cc5eb 235fca383cb5ab9d892faf3608489cd56655ed87 50771e305929168d44091e11e7491da35559a9b4 be4c1c202a2583d9d49a8e9881f9f57b37ae16b2 1e2575e266514625ca7591abd6807877842b91c1 1d9d53f16af5ea60fef032102f710f0a2f0a098a 3534182525c52ccec2f508d8664dc90899c54c6d 3b6805317c906a98d34c3d483e720bccfb138922 b47e7221fcee6b995728eec97b093e7a201990c9 69a09fbb98d0f8679f0717f3bdbf32ae24bf69cc 53304574aa944b6696c44f76f8ce45f75f3cf641 a32a0faeeec65891f93157bc368dea54383f6d89 381b398188662098c306121a92c7164b753eb1ee 7a9f8f0e7a443ac2ed5d7dab4e5f9568a85c7479 3c4b3dd2375681a3d59d1ea2af7deb8cdf4c6990 e3df83d9b798be3284194bb4390579ac7d196873 393580b1980ac00490dc8e14c68091e8ac5694ad afa991896b637884e7d60aaf6024a6457c96c0b0
detroie	c9d4d366439d144fdb92c60562cda4c9be0e0b04 61e7feb76f046262e7b8ebd08450e50c4cef6ff1 801c0bd52c63d8aa17491e4e7f017435314ae360 62c42f510f93dd0ce86d5666449ac9d733b15e9a 576a946cdfa6feadf06142ae06f55b7fed577115 09031d53c01585fc2a1ff8358cb0541d2006e799 3ab89751d14866299b2cb78eb41f931adc816717 1aed3dd03d42b96a376598c7a53b5edab14e69f9 e6fb750288954e94917a0f7a2ee7b5b271672885 6abb9084345b9e2990e14fe907e6fd213cc47cc4 e9abd5e7bf0d77723991df905615d14dd4e31692 2cad3027ce9674f00cdfd7487e83956033dee1d8 2739c2937b4de40924a0400a2200caa9f271f51d 59037dc39c4a18d6b98280a9e598c83a33c3e64b 9917463fd83f55c5afe4cd5fa5dec1ef8e2128e2 13029bb8fd12b4ec54dc49ea49a90abdc2fc75ee 04226396c0de4dce851f7529f88fdb96dd112d57 6fc2c149f385fb2c35ade0b0dcea54c7ec4b9422 97d69adc224831a186c5036e0985ce991177dae9 bf63e4ae6b88b1fa3b7426adaf6e1ecbf6c9af4a 7f6c111ac542f49c83a3a90514497f0ede8b7392 2834829c99ef5cafcbcdfe0f5c194f3f0c096ad1 bd287cfe53be4f13404d9c081db5cc223ab380ad 6a63611845806950c9f3f03f9dc8adb660aae4a0 144837eb6d598644ae5664eadaa3efe308d4a2cd

Table A.7: Evaluation dataset (4/5).

Family	SHA1
cleanware	168518b3b4aa0ae5c9ade193177ff1a0125ec665 c00a10899c17f66ad4b47d9b2e57a9cef4004b6d 934ce1475dcaff413a60d792143a582064a62f5b a1bf405584a3b93912c8ae0fb1aa137b4a1f387c 0b1d02a4d87c2780dd3d8958cfe9b9b6757bc71d 43b18970cb4571b8b516ec4eab5dcba6517b4dd8 1ab2d93096bcc18c6956118860c3e9aaa4c2f8cf cdf15ab96e7d9e50bd98d20564b67bfa4113c8fb 976d42ff400556a15946c1bad7d687212d0c96b3 cfba36ab1f6a8842cfa405e922e6c261ac2b45a2 60e870128b89da49334f107e289d7a74eef7ccad 62ec427462f477ff810198aefd122af73a68c73c b8abcc65de8e8f8edc4b27569b34d10bb8c13e4d ee9847051a898d88060ca52bab0c229851acfbf0 eed8781c92830087cc2cc41b507b7b6899ae8f17 9adf02ba06b5297d95e8288cc50ae44ec9f8c28f a51959cb4e11467f0fbdf490be2078ff5d9d7391 b63b81aa89fd8db3d9ff2386cc816d5849d95869 39a6e114bc4f4bcaf9c915abc5158c98fa2ab4da 1898cd90ad7095121511383170d5b3ee85325448 139926599045c7de0b37c0906b51c0322c9ab82a f36ce7091903b73a6905460069877ddc209ad2e7 e0a85bee630d831e3f1aa99bf87b2d7b379f1b40 50260b3c03863a375f226c6e1e124b93c325c12e 3b5ce23f1e64cb41f639b68fddc44a51f871f70a

Table A.8: Evaluation dataset (5/5) - Cleanware.