Composite Software Diversification

Shuai Wang, Pei Wang, and Dinghao Wu
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802, USA
{szw175, pxw172, dwu}@ist.psu.edu

***Abstract*—Many techniques of software vulnerability exploitation rely on deep and comprehensive analysis of vulnerable program binaries. If a copy of the vulnerable software is available to attackers, they can compose their attack scripts and payloads by studying the sample copy and launch attacks on other copies of the same software in deployment. By transforming software into different forms before deployment, software diversification is considered as an effective mitigation of attacks originated from malicious binary analyses.**

Essentially, developing a software diversification transformation is nontrivial because it has to preserve the original functionality, provide strong enough unpredictability, and introduce negligible cost. Enlightened by research in other areas, we seek to apply different diversification transformations to the same program for a synergy effect such that the resulting hybrid transformations can have boosted diversification effects with modest cost. We name this approach the *composite software diversification*.

Although the concept is straightforward, it becomes challenging when searching for satisfactory compositions of primitive transformations that maximize the synergy effect and make a balance between effectiveness and cost. In this work, we undertake an in-depth study and develop a reasonably well working selection strategy to find a transformation composition that performs better than any single transformation used in the composition. We believe our work can provide guidelines for practitioners who would like to improve the design of diversification tools in the future.

***Index Terms*—software diversification; reverse engineering; binary instrumentation;**

I. Introduction

With the rapid development of software reverse engineering and analysis, attackers have gained a certain level of advantages in the arms race. Code-reuse attack analyzes the victim programs to identify sequences of reusable code snippets and direct the control flow through these snippets to construct malicious operations [1], [2], [3]. Patch-based exploitation analyzes the post-patch binary code to expose hidden vulnerabilities (fixed by the patch) and construct attacks towards the pre-patch binary [4].

Software diversification produces different variants of a program, altering software syntax but retaining the semantic equivalence. After diversification, each copy of the software has a different structure. Therefore, knowledge obtained by reverse engineering one copy of the software is not applicable to other copies, making attacks depending on such knowledge (e.g., code-reuse attack and patch-based exploitation) lose generality or not feasible at all.

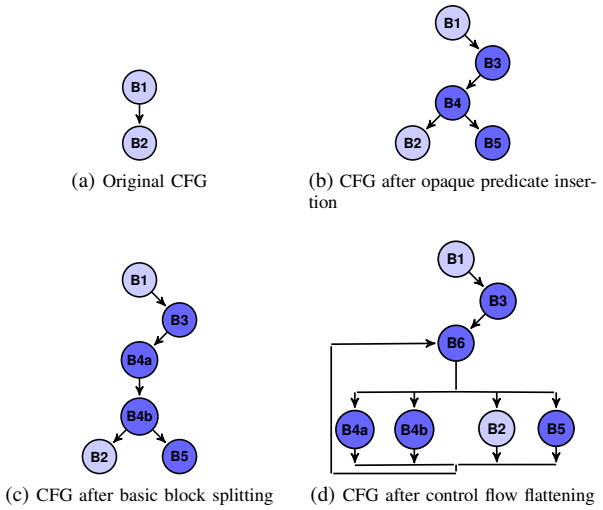
There has been many great work on software diversification and a large portion of them focus on the transformation algorithm side [5], [6], [7], [8], [9]. A good transformation algorithm can vastly mutate the binary form of a program with a considerable amount of randomness. Meanwhile, the transformation preserves the original semantics and keeps the incurred cost as modest as possible. Typical penalties of diversification transformations include binary size expansion and execution slowdown. As more and more transformation algorithms have been proposed, it becomes more and more difficult to develop new algorithms that provide reliable diversification effects with low cost. We have noticed that recent progress on software diversification is more about building frameworks and providing support for upstream techniques (binary rewriting, for instance) which enable software diversification in different scenarios [10], [11].

In this research, we propose *composite software diversification* which combines existing methods together for a synergy effect. The basic idea is that by applying different diversification transformations to the same program, we can make the binary more efficiently diversified compared to applying a single transformation; meanwhile, the cost of the composite transformation is kept low enough for practical deployment. The composite diversification, if feasible, can extract the hidden value of past research results and greatly enrich the choices of software diversification algorithms.

The idea of combining different program transformations of the same kind for greater benefits is not merely an intuition but has been proven reasonable and feasible by previous work on compiler optimization [12]. Although optimization and diversification have different goals and are evaluated with entirely different metrics, we do believe that the success of the idea in one area gives a strong hint that similar methods can work in another field.

Given a set of primitive program transformation algorithms, the search space for an optimal or a close-to-optimal composition is considerably large. To investigate the feasibility and effectiveness of composite software diversification, we propose a methodology that comprehensively evaluates a diversification transformation, either primitive or composite. With this methodology, we further develop a strategy to prune the search space so that our study can be done in an empirical way; this strategy itself has a reference to the data mining research. To the best of our knowledge, despite the growing need for deploying diversified real-world applications, no systematic study has focused on comprehensively evaluating the performance of software diversification transformations when they are composed together.

In summary, we make the following contributions:



(a) Original CFG (b) CFG after opaque predicate insertion (c) CFG after basic block splitting (d) CFG after control flow flattening

Fig. 1: Program diversification in multiple iterations.

- We propose a new concept of software diversification called *composite software diversification*. By composing different diversification transformations in a certain way we can boost the effectiveness of the previously proposed methods while keeping the cost of diversification under control.
- By referring to data mining research, we develop a fairly well-performing search strategy. This strategy effectively selects the satisfactory composition of transformations which diversifies a program without incurring much cost.
- We justify our research idea and methodology with extensive empirical experiments on the C programs of the SPEC2006 benchmark suite. The results show that composite diversification is a promising technique that should be appealing to software developers and distributors.
- We developed a tool called AMOEBA, which delivers composite software diversification to binary code (it is publicly available for download at https://goo.gl/UaMRzS). To our best knowledge, there is no publicly available diversification tool on binary code, and we contribute to filling the gap.

II. Composite Software Diversification

In this section, we first present a motivating example that demonstrates the synergy effect of composite software diversification. We then formalize the problem that our research is to solve.

A. Motivating Example

The key observation is that a program can become more *diversified*, by composing multiple transformations on the processed binaries. We use a simple two block control flow graph to illustrate our observation. As shown in Figure 1a, the two blocks are connected by a direct `jmp` instruction. The original control structure is firstly transformed through an opaque predicate [13]. As shown in Figure 1b, `B1` will have to invoke function `B3` to get the predicate. A conditional `jmp` is implemented in `B4` to check the predicate, and if it fails, `hlt` instruction will be executed in `B5`. The `B4` is then transformed by splitting itself into `B4a` and `B4b` via a inserted `jmp` instruction, as shown in Figure 1c. Finally, the output CFG is transformed by `control flow flatten`. As shown in Figure 1d, intra-procedural control flow graph is flattened (the graph in Figure 1d is simplified). Instructions are inserted to update each control flow transfer destination to a global variable, and the control transfers are redirected to `B6`. An indirect `jmp` in `B6` uses the address in the global variable to transfer the execution flow. Note that after three transformations, the 2-block graph is extended to a 7-block graph with a more complex structure. In summary, by reprocessing the output with different transformations, the diversified code can become progressively complex.

B. Problem

A program transformation could be formalized as a function $T : \mathcal{P} \rightarrow \mathcal{P}$ where $\mathcal{P}$ is the universe of all programs. In software diversification, T may be probabilistic, meaning the output of applying T to the same input is not unique but follows a distribution specific to T. In the rest of the text, we always consider diversification transformations.

Given two transformations T_1 and T_2, a mixture of them could be the composition $T_2 \circ T_1$, namely we first apply T_1 to a program and then apply T_2 to the output of T_1. Just like the expectation we have for any hybrid method, it would be ideal if $T_2 \circ T_1$ outperforms both T_1 and T_2 applied alone, concerning the given criterion for measuring the performance of software diversification. If $T_2 \circ T_1$ indeed has the better performance, we say there is a synergy effect between T_1 and T_2. The same study can be done with $T_1 \circ T_2$ which is another way to compose T_1 and T_2.

In this research, we would like to investigate if such synergy effect generally exists, especially when there are more than two primitive transformations to compose. If the synergy exists indeed, we want to develop a method that can effectively and efficiently achieve it. Since the synergy may manifest only if the transformations are composed in a particular way, this problem grows beyond trivial as the number of applicable primitive transformations increases. Therefore, the first step is to make a clear definition of the problem by generalizing the previously made example.

Suppose we have a set $\mathcal{T}$ containing k diversification transformations $T_1, \cdots, T_k$, we can construct the set of all possible compositions in the following way.

$$C_n = \{T_{i_n} \circ \cdots \circ T_{i_1} \mid \forall l \in \{1, \cdots, n\}, T_{i_l} \in \mathcal{T}\}$$
$$C^* = \bigcup_{n=1}^{\infty} C_n$$

Therefore, the objective of our research is to develop a searching strategy which can find a subset of C^* such that the composed diversification transformations in this subset have the optimal or close-to-optimal performance under certain evaluation criteria.

III. Experiment Setup

Since there are very few mature formal theories on program diversification, we try to solve the problem in an empirical way. That is, by actually implementing a set of composite

diversification transformations and developing a set of measurements we compare the performance of different compositions with field experiments. We believe that with carefully designed experiments and evaluation metrics, empirical results can still have a certain level of generality even without strong theoretical foundations.

From the construction of the problem space it can be seen that the choice of three factors decides a composition, which are 1) the length of the composition ("length" refers to the number of iterations), 2) the subset of primitive transformations to use, and 3) the order in which these transformations are composed. Apparently, enumerating all choices for the three factors is not feasible, so the major challenge of this research is how to reasonably prune the problem space so that the empirical evaluation can be done with limited resources.

In this research, we focus on the first two factors. Given the choices of the first two factors, we randomly decide the third factor to construct a composition. There are two reasons behind this decision. The first is that among all three factors, the third one expands the problem space most vastly. The second reason is that randomizing the sequence of composition potentially makes the diversification more unpredictable, which is a significant benefit in practice. Nevertheless, heuristic-based approaches, or perhaps machine learning-based approaches could facilitate the study of the third factor effectively. We leave it as one future work to investigate the third factor in boosting composite diversification.

Note that software diversification can be performed at different stages, e.g., at the pre-compile time, compile time, or at the post-link time (i.e., binary retrofitting). Although composite diversification in general is applicable at any of these stages, we choose to implement AMOEBA and demonstrate it with post-link time diversification because security hardening through binary retrofitting does not require source code and therefore has a broader application scope. We leave it to software developers to decide at what stage to perform composite diversification in the real-world scenarios.

The underlying reverse engineering facility AMOEBA relies on is an open-source disassembler called UROBOROS [14], [15]. Starting from the assembly code of the original binary, we iteratively apply different diversification transformations to the program, producing a unique variant each time. All transformations we use in UROBOROS are from existing software diversification research that are relatively *simple* and *straightforward*. The overlay of different simple transformations eventually leads to a synergy effect after a particular number of iterations, making the produced variants resilient to certain security threats and binary similarity detection.

A. Diversification Passes

We intuitively choose ten "classic" binary diversification methods proposed by existing research as the transformation candidates. In the rest of the paper, we name each transformation as a *diversification pass* applied to input programs. Most of these transformations have also been indexed by an influencing literature review on software diversification [5]. The roster of these transformations is in Table I. Note that we use these simple transformations in our study, but more advanced and sophisticated transformations can be implemented as well. There are three levels of assembly transformations—instruction level, basic block level, and function level. At this point, it is still unknown whether each pass in Table I will be used in composite diversification or not. An in-depth selection process is required to decide an appropriate combination of passes for effective diversification. Pass selection will be discussed later in Section IV. We now elaborate on each pass.

TABLE I: Diversification pass candidates.

Class	Methods
Instruction Level	instruction replace [6] instruction insert [6]
Basic Block Level	basic block reorder [7] basic block merge [5] basic block split [5] opaque predicate insert [13] control flow flatten [13] branch function insert [8]
Function Level	function reorder [9] function inline [5]

Basic Block Reorder. This diversification rearranges the relative positions of basic blocks. In particular, two basic blocks are randomly selected as candidates to reorder, with necessary control transfer instructions and labels inserted in the context to preserve the correct semantics. For each pass, we reorder one pair of basic blocks from each function.

Basic Block Split. This diversification splits one basic block into two by inserting a `jmp` instruction in an arbitrary position and set its destination to the associated next instruction. We randomly select one basic block to transform from each function.

Basic Block Merge. This transformation searches for mergeable basic blocks. A basic block is defined as mergeable if it has only one predecessor, and its predecessor has only one successor. For each pass, we randomly select one pair from all the mergeable basic block pairs and merge them together.

Instruction Insert. This diversification inserts meaningless code sequences into the program. For each function, we randomly select one basic block and insert a sequence of garbage instructions with a random length (3–5). The insertion candidates are `nop`, `mov %esp, %esp`, `lea 0x0(%esi),%esi`, and `xchg %esp, %esp`.

Instruction Replace. This strategy searches for typical instructions and replaces the targets with its semantic equivalent substitutions. We adopt two substitution strategies to transform `call` and `ret` instructions. When replacing `call` instructions, `jmp` instructions are used to transfer the control, and the return address is explicitly saved on the stack by a `push`. `Ret` instructions are replaced in a similar way, with a `pop` instruction to obtain the return address on the stack.

Control Flow Flatten. As shown in Figure 1d, this transformation flattens the control flow graph. Given a target control structure, control flow transfers are redirected to a dispatcher block inserted by this transformation. The dispatcher leverages a global variable to decide which block to jump, and instructions are inserted at the end of each basic block to update the global variable with control destinations. For each pass,

we randomly select one function and flatten its control flow graph for each iteration.

Opaque Predicate Insert. As shown in Figure 1c, a basic block `B` can be guarded with a conditional branch to `B` and another (garbage code) block `B'` using an arbitrary predicate. A `call` instruction to the predicate function and a conditional jump instruction are inserted at the beginning of target block. For each pass, we insert one opaque predicate for each function.

Branch Function Insert. This transformation substitutes `jmp` instructions with `call` instructions to the branch routine and the `jmp` destinations are updated into an artificial global variable. The branch function utilizes an indirect `jmp`, transferring to the destination stored in a global variable. We transform all the identified candidates.

Function Inline. This transformation inlines functions into their call-sites. Direct `call` instructions to the target function are found and the target function is inserted after these call-sites. The `call` instructions are changed into `push` and `jmp` instructions to simulate the original semantics. `Ret` instructions in the target are also rewritten into `jmp` instructions. As destinations of an indirect function call are hard to analyze, we conservatively leave the target function in its original place. In the implementation, one function is randomly selected to transform as long as its size is less than a predefined threshold (the threshold is set as 500 bytes in this paper).

Function Reorder. Same as basic block reorder, this transformation rearranges the relative positions of two functions. In case the execution flow falls through the function boundaries, we insert `jmp` instructions and corresponding labels in the predecessors and successors of the reordered pair, thus delivering the equivalent semantics. For each pass, we reorder one pair of functions.

B. Measurement

The goal of composite diversification is to provide production of low cost and well-diversified software variants. To assess our fulfillment of this objective, we need to quantitatively measure cost and diversity.

1) Cost: We assess cost with two metrics—size expansion and execution slowdown of diversified binaries. The cost is a concern in composite diversification because most passes in Table I insert new instructions or new control flow transfers into the binary, which will inevitably affect the size and speed of the products. We leverage `bzip` (§IV and §V-A) and SPEC2006 programs (§V-B) in our experiments. The execution speed (i.e., slowdown) of diversified variants are measured through the test cases shipped with the programs. The size is calculated using the `stat` program from GNU Coreutils. Our experiments are launched on a machine with Intel E5-2690 2.90GHz with 128GB memory running Ubuntu 12.04 LTS.

2) Diversity: Another aspect of the assessment is to measure the diversity of the variants produced by composite diversification. We present our quantitative evaluations on diversity regarding two typical threats that software diversification can hinder, i.e., code reuse attack [3] and patch-based exploitation [4]. The security strength of a diversification method can be well reflected by its resilience to these two adversaries.

Resilience to code reuse attacks can be evaluated by the elimination rate of return-oriented programming (ROP) gadgets. ROP attack is one state-of-the-art program exploitation which manipulates program call stacks and chains sequences of victim program's own code snippets (named ROP gadgets) to perform arbitrary operations [3], [16]. A general assumption made by related work is that attackers need to know the memory addresses of ROP gadgets in order to tamper the call stack [11], [5], [10], [17], [18], and if a ROP gadget changes its location or no longer exists in the diversified binary, attackers will have difficulty in reusing the existing attack payloads. We use the ROP gadget harvesting tool ROPGadget [19] to search for gadgets in binaries. Gadget elimination rate between two binaries is defined as

$$1 - \frac{|A \cap B|}{\min\{|A|, |B|\}}$$

where A and B are the sets of gadgets found in two binaries. Two gadgets are considered equal if they have the same instruction sequence and starting address. Note that recent research work has proposed advanced methods to launch ROP attacks without the pre-knowledge of ROP gadgets (i.e., Just-In-Time ROP attack [20], [21]). We present our further discussions regarding this topic in §VI.

Resilience to patch-based exploitation can be evaluated by investigating how well the diversified binaries can mislead binary diffing tools [22], which can be used to locate the vulnerability by comparing the semantics of the original binary and the patched one. We use `BinDiff` (version 4.0.1) [23], the de facto industrial standard binary diffing tool available on the market, to calculate the similarity between two binaries. Given two binaries, `BinDiff` provides the number of matched functions, basic blocks, and instructions. Since function and basic block can be "partially" matched, e.g., 30% of the instructions in a function are matched with another function, counting the number of matched functions or basic blocks could be tricky. Therefore, we only adopt instruction matching rate. The rate is calculated by

$$\frac{|A \cap B|}{\min\{|A|, |B|\}}$$

where A and B are the sets of instructions in two binaries. Two instructions are considered equal if they are matched by `BinDiff`, so $|A \cap B|$ is the number of matched instructions. It should be noted that being semantic equivalent does not necessarily make two instructions a match; `BinDiff` also takes the contexts of the instructions into account [24], [25]. `BinDiff` does provide an overall similarity score to summarize the comparison. However, it is unclear how this score is computed. To make our results more interpretable, we do not use it in our evaluation.

Unlike cost assessment which only compares every diversified binary with the original one, evaluation on the diversity of composite diversification needs an additional step. Since attackers are usually not limited to only chose the original binary to analyze, the diversity of variants should reach the **pairwise** granularity. That means, every pair of the generated variants should be different enough so that attackers cannot exploit any other variant by reverse engineering one of them.

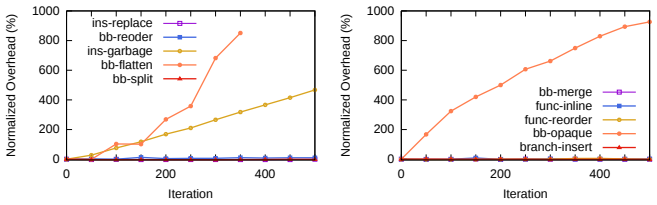


Fig. 2: Execution slowdown by single-pass diversification.

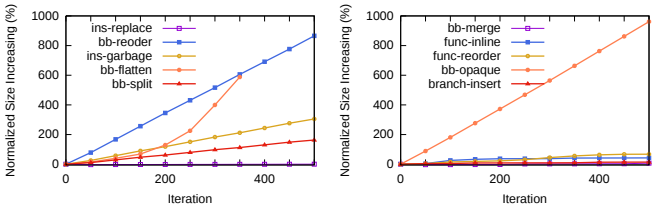


Fig. 3: Size expansion by single-pass diversification.

IV. Pass Selection

Given the candidate passes in Table I, we want to find an applicable subset of them as the primitive diversification transformations to employ in composite diversification.

A. Selection Methodology

Having decided the metrics used for assessing the performance of composite diversification, we can start searching for the subset of diversification passes that can be employed by our implementation of composite diversification. Given 10 passes, there are a total of 1023 different non-empty combinations of them if we do not fix the number of passes to pick. Assessing all possible combinations is unlikely to be feasible.

To address this issue, we propose a two-stage pass selection method. In the first stage, we evaluate the cost of every single pass when they are repeatedly applied to a binary for many times. After this first-stage selection, passes that are too costly in the context of composite diversification will be ruled out for further consideration. Hopefully, the first-stage selection can reduce the total number of passes we need to consider in the second stage.

We leverage program `bzip2` (version 1.0.3), a widely-used data compressor as the experiment object in the selection process. For each diversification combination, we iterate it for 500 times, which we believe is significant enough for an in-depth study. These 500 iterations lead to 500 variants, each of which is based on the previous one instead of the the original. Before launching selection steps below, we first verify the functional correctness of these diversified outputs using the test cases shipped with `bzip2`. We report all the outputs can pass these shipped test cases.While the adopted algorithms are supposed to produce equivalent code, we test the functional correctness to confirm the faithful implementations of these algorithms in Amoeba.

When comparing the diversified binaries with the original one, we do a 1-in-50 sampling on the 500 variants, leading to a sample size of 10. For pairwise comparison on diversity-related metrics, we randomly pick 50 pairs of variants.

B. First-Stage Selection

The first-stage selection measures the singular cost of each diversification pass. We consider a pass to be too costly if the size expansion grows super-linearly, or the execution slowdown does not grow sub-linearly with respect to iteration times because users are usually much more sensitive to execution speed than binary size increases.

Figure 2 and Figure 3 show the size expansion and execution slowdown of diversified `bzip2` variants over 500 iterations, for all 10 diversification passes. While most of the transformations only introduce negligible runtime overhead, the impact of basic block flatten, instruction insert, and opaque predicate insert is out of the scope of our tolerance for execution slowdown. As for size expansion, the increasing trends of most passes are linear, except basic block flatten. According to our definition of costly transformations, basic block flatten, instruction insert, and opaque predicate insert will be excluded from further consideration by our composite diversification framework, reducing the number of candidate passes from 10 to 7.

TABLE II: Candidate diversification pass combinations generated by backward-stepwise selection (shaded column indicates the best of all).

	mix0	mix1	mix2	mix3	mix4	mix5
Instruction replace	✓	✓	✓	✓		
Basic block reorder	✓	✓	✓	✓	✓	✓
Basic block merge	✓	✓	✓	✓	✓	
Basic block split	✓	✓	✓	✓	✓	✓
Branch function insert	✓					
Function reorder	✓	✓	✓			
Function inline	✓	✓				

C. Second-Stage Selection

Although the first-stage selection has pruned a few passes, the remaining search space is still too large for us to enumerate. Therefore, we need a strategy to further compress the pass selection process in the second-stage.

After referring to previous research and related disciplines, we decide to borrow a selection method from data mining. There is a classic problem in data mining called regression which seeks to estimate the relationship between a response variable and a set of predictor variables. In regression, a mathematical model with configurable parameters is assumed, from which the response can be computed based on the values of predictors. However, it is common that only a subset of the predictors are actually related to the response, so regression has to decide which predictors should be selected for fitting the model. Similar to our situation, enumerating all possible combinations of predictors usually needs unaffordable resources. Therefore, data mining researchers have developed numerous predictor selection methods to avoid brutal-force search for an optimal model.

In this research, we employ the *backward-stepwise* method [26], [27] for pass selection. In backward-stepwise selection, the baseline is first set as the model containing all candidates (predictors in data mining and diversification passes in our case). Starting from this baseline, the selection process generates a set of new models by removing one candidate from the

baseline model. Among all newly generated models, the one with the best performance is picked as the new baseline. The selection process repeats in this way until the latest baseline model consists of only one candidate. For our problem, we stop when the baseline model has only two passes, because we have already assessed the performance of every single pass. When the selection process is over, each baseline model is considered as the best model among all models with the same number of candidates. The final step would be comparing all baseline models.

There is a similar method called forward-stepwise selection. The difference is that instead of eliminating candidates from the baseline model one by one, forward-stepwise gradually adds new candidate starting from the empty model. Compared to backward selection, forward selection tends to generate a model with fewer candidates. Note that in composite diversification, we want to *maximize the diversity* of the passes we use in the framework, so we choose to build the model backward instead of forward.

Apparently, there is no guarantee that stepwise selection gives the globally optimal combination; however, the method remains one of the most widely used because of its simplicity and fairly good performance in practice.

When comparing two diversification plans for large iterations, it is hard to give a particular criterion that will be universally applicable. Depending on the characteristics of the program to protect and the demand of users, the comparison result could be different. We do not try to develop a versatile comparator for evaluating diversification plans. Instead, we only propose a reasonable comparator to illustrate the feasibility of our proposed framework.

In this work, we assume users care about execution slowdown most, so it will be the decisive factor when deciding the best diversification plan. On the other hand, we do not want a winning plan that suffers from some obvious drawbacks. For that purpose, we design the comparison method in a filter-oriented manner. A plan is discarded if it does too badly with respect to any one of the metrics mentioned in Section III-B. After the filtering is over, we pick the best plan based on the metric we care about most, which is execution slowdown in our illustration.

A plan is considered to be too poor at a metric if its score on that metric is an outlier in the undesirable direction among the scores of all plans. In statistics, a data point is called an outlier if it is greater than $Q_3 + (Q_3 - Q_1) \times 1.5$ or smaller than $Q_1 - (Q_3 - Q_1) \times 1.5$, where Q_1 and Q_3 are the first and third quartile. For example, if the execution slowdown of some plan is an outlier at the high end, we will filter out that plan because high execution slowdown is unwanted. On the other hand, if the gadget elimination rate of some plan is an outlier at the low end, we will also filter it out because low gadget elimination rate indicates poor diversity.

Note that the filter-based selection may result in a situation where all candidate plans are pruned. That would mean every plan has at least one weakness. Any of such situations happening would threaten the rationality of our selection method. Nevertheless, none has manifested in our experiments.

TABLE III: Mean of metrics used for plan selection (shaded cells are outliers).

	mix1	mix2	mix3	mix4	mix5	Norm. Range
Slowdown (%)	3.42	-0.81	0.45	-0.31	3.20	[-5.59, 8.47]
Matching (%)	16.08	16.04	14.96	14.90	86.89	[13.28, 17.76]
Pair.Match. (%)	47.25	35.72	34.88	34.70	38.84	[28.94, 44.78]
Elim. (%)	98.21	98.21	98.22	98.22	98.22	[98.20, 98.24]
Pair.Elim. (%)	98.01	97.97	98.12	98.29	98.46	[97.59, 98.71]

Due to limited space, we are unable to show the analysis result of all combinations that have appeared in the selection process. We only present the final selection, i.e., selecting the best baseline model. Recall that the baseline models are the best-performing combinations in each round of backward-stepwise selection. We list these combinations in Table II.

To illustrate the selection process, we first present the performance data of the diversification plans. We measure all metrics for mix1 to mix5. mix0 only has data reflecting execution slowdown. Since its runtime overhead is clearly unacceptable for composite diversification (up to 462% after 500 iterations), there is no need to consider it in subsequent selection.

Figure 4 shows the execution slowdown of each combination. As can be seen, the overhead of mix0 is significantly higher than the rest, while overheads of other plans are considered sub-linear (satisfying for further study). Figure 5 shows the size expansion evaluation. All the plans show roughly linear expansion, which is quite consistent with the size evaluation in the first-stage selection (§IV-B). For diversity between the generated variants and the original binary, Figure 6 and Figure 7 show the ROP gadget elimination rate and instruction matching rate from `BinDiff`, respectively. As shown in Figure 6, almost all the ROP gadgets are eliminated after transformations; we report on average 98.2% gadgets become un-reusable. Binary diffing evaluation also shows promising results. Actually besides `mix5` (which will be filtered out due to the high remaining similarity), all the plans show notable decrease in the instruction matching rate. We also present the pairwise diversity evaluation in Figure 8 and Figure 9. Pairwise ROP gadget elimination rate shows promising results; we observe stable high elimination rates for all the compared pairs. As for the pairwise instruction matching, we report `mix1` has relatively high remaining similarity (47.3%), while the other four plans show much better results (on average 36.0%).

Table III summarizes the experiment data by computing the average of each metric. As previously described, we pick the plan with the lowest runtime overhead, after filtering out plans with salient weaknesses. The normality range of each metric is computed, also listed in the table. As can be seen, mix1 is pruned due to lack of pairwise diversity; mix5 is filtered out because of high similarity between the variants and the original binary. mix2 is selected as the best diversification plan because it has the lowest execution slowdown and does not suffer from any significant weakness.

D. Multi-Pass Versus Single-Pass

After the two-stage pass selection, we have chosen mix2 as the winner combination of transformation passes for composite

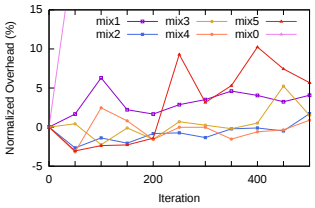


Fig. 4: Execution slowdown by multi-pass diversification.

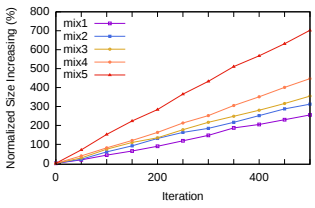


Fig. 5: Size expansion by multi-pass diversification.

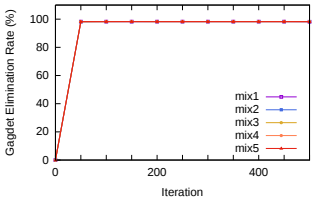


Fig. 6: ROP gadget elimination by multi-pass diversification.

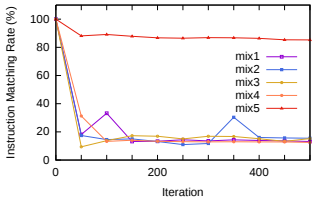


Fig. 7: Binary diffing by multi-pass diversification.

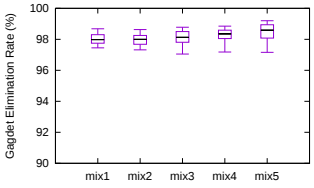


Fig. 8: Pairwise ROP gadget elimination by multi-pass diversification.

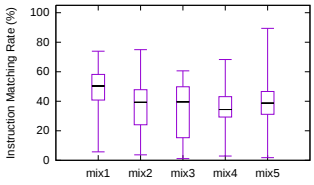


Fig. 9: Pairwise binary diffing by multi-pass diversification.

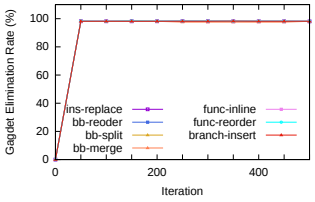


Fig. 10: ROP gadget elimination by single-pass diversification.

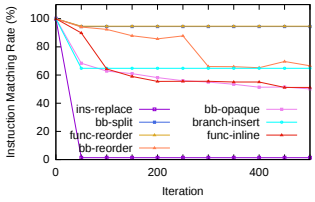


Fig. 11: Binary diffing by single-pass diversification.

diversification. However, we have not yet showed that mix2 can outperform single-pass diversification in terms of diversity. To prove that the synergy effect we have discussed in Section II does exist, we launch a competition between multi-pass and single-pass diversification. Measurement on the diversity of single-pass generated variants is illustrated in Figure 10, 11, 13, and 12.

We first make a comparison between single-pass diversification and multi-pass diversification on their resilience to ROP attacks. Figure 10 and Figure 6 suggests that there is

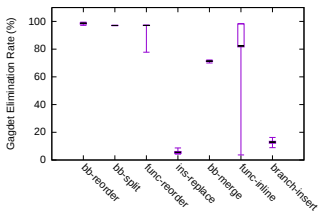


Fig. 12: Pairwise ROP gadget eliminate by single-pass diversification.

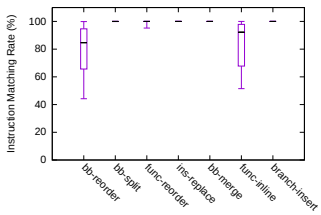


Fig. 13: Pairwise binary diffing by single-pass diversification.

no significant difference between single-pass and multi-pass plans on ROP gadget eliminate rates when comparing the variants with the original binary. In the pairwise comparison showed in Figure 12 and 8, however, the performance of some single-pass plans is clearly inferior to multi-pass plans. For example, instruction replace has nearly zero pairwise diversity. The reason is that after the first round of diversification, there are no suitable instructions for this pass to replace, making it an idempotent transformation. Differently in multi-pass diversification, the collaborating passes (basic block reorder, for instance) keep inserting instructions that can be replaced, making instruction replace an efficient pass throughout the iteration.

In the competition on resilience to patch-based exploitation generation, the advantage of multi-pass diversification is even more significant. When matching the diversified binaries with the original (Figure 11 and 7), most multi-pass plans can reduce instruction matching rate to about 20% after 50 iterations. In contrast, matching scores from most single-pass plans stay above 45% even after 500 iterations. The only exception is instruction replace, which can achieve nearly 0% matching score. Our guess is that replacing `call` and `ret` with `jmp` causes exceptional hardship for `BinDiff` when it tries to recover the control flow, which is the crux of its matching algorithm. Nevertheless, instruction replace is an idempotent pass, meaning it will surely have poor performance on pairwise matching. Actually, Figure 13 and 9 show that all multi-pass plans provide much more pairwise diversity than single-pass plans. At this point, we have enough evidence to conclude that multi-pass diversification is more effective than single-pass diversification. This conclusion further justifies the election of mix2.

V. Validation

In this section, we validate our approach in two aspects. We first compare our backward stepwise selection against a baseline approach, i.e., random selection (§V-A). We also validate our optimal combination with multiple large size programs from the SPEC2006 test set (§V-B).

A. Comparison with the Baseline Method

In our pass selection step (§IV), we initialize our selection from ten widely-used program diversification methods. Since the ordering between different techniques are not considered

(§III), ten methods lead to 1023 non-empty combinations. After the first phase, 3 of the methods are eliminated, resulting in 127 different possible combinations. Our tentative tests show that it takes a non-trivial time to apply transformations for hundreds of iterations. Thus, running diversification for all 127 combinations and pick the best available option cannot be done in a reasonable amount of time. As presented in §IV-C, by using stepwise selection to find the optimal combination, we have to test 27 different candidates (7+6+5+4+3+2). In this section, we study whether a random selection approach (i.e., the *baseline method*) can find a better combination in 27 different runs.

We randomly select 27 combinations from 120 possible combinations which contain *at least two methods*. Our study on the `bzip2` program has shown that after 100 times, there is no significant benefit for most of the methods regarding binary similarity (Figure 7). Thus, the process is iterated for *100* times for each combination. We evaluate the performance and cost of the 100th diversified output; we also compare the 100th and the 50th diversified outputs for the pairwise metrics (in this step, experiment results of mix2 are acquired in the same way). The optimal combination from 27 candidates is selected regarding the same filter-based selection strategy employed in §IV-C.

We launch this 27-round random process for 20 times, resulting in 20 control groups.[1] We now report the key observations. In general, all 20 control groups show comparable execution slow and size expansion with mix2. On the other hand, while binaries from control groups has acceptable (pairwise) ROP gadget elimination rate and low similarity rate, we observe 18 control groups suffer from unsatisfying *pairwise* similarity rate. In particular, test on mix2 reports a low pairwise similarity rate (around 40%), while 18 control groups have over 95% pairwise similarity rate. There are only two well-performing control groups, i.e., group6 and group10. We report that group6 has the exact same combination with mix2, while group10 has a combination of three methods, i.e., instruction replacement, basic block reorder and basic block split. Overall, our study shows that for these 20 control groups, only two of them show comparable performance with mix2, and there is no control group can notably outperform mix2. We interpret this as promising results to show our stepwise selection can quickly construct optimal combinations with reasonable amount of effort.

B. Test on SPEC Programs

In Section IV we have selected a set of diversification passes as the primitive transformations to use in composite diversification, based on experiment data on a single program `bzip2`. In this section, we validate our winning diversification plan mix2 on a larger set of programs, i.e., all C programs in SPEC2006. For validating the cost and effectiveness of the diversification plan, the experiment setting and metrics to measure are same as the selection process. Although our experiments on `bzip2` indicate that there may not be obvious benefit after 100 iterations, the SPEC programs are still conservatively processed for 500 iterations. We hope this *extreme* setting can better reveal the advantage and limitations of our technique and facilitate software developers with detailed information regarding real-world deployments, while a user can choose the needed number of iterations in practice. Same as §IV-A, before launching experiments below, we first verify the functional correctness of all the diversified outputs. We report all the outputs can pass the test cases shipped in SPEC2006.

[1]The full comparison can be viewed at https://goo.gl/3XbtVR.

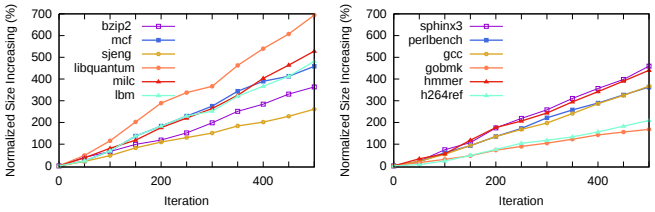


Fig. 14: Size increase for SPEC binaries.

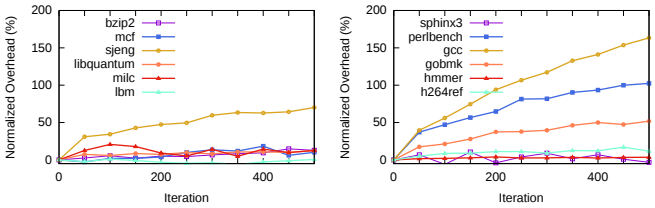


Fig. 15: Runtime overhead for SPEC binaries.

The size expansion of binaries used for validation is given in Figure 14. The data shows that the augment of binary size is bounded by 700% in 500 iterations, for all tested programs. Moreover, the trend of size expansion is linear with respect to original binary size and number of iterations. We believe this amount of cost is acceptable, at least for desktop and server computing environments.

The execution slowdown of the diversified binaries is probably more of a concern for composite diversification. The trend of runtime overhead increase over iterations is presented in Figure 15. According to the graph, overhead of all programs grows sublinearly, which fits our objective.

For diversity validation, Figure 16 presents evaluation on the ROP gadget elimination. Almost all the ROP gadgets become unavailable after transformation. We also report besides three test cases which have a relatively high instruction matching rate (`bzip2`, `lbm`, `mcf`), average matching rate of all the other test cases are less than 15% (Figure 17).

Figure 18 and 19 present the pairwise diversity. While all the pairwise gadget elimination tests show promising results (on average 97.51% gadgets are eliminated), we observe one outlier (`perlbench`) in the pairwise binary diffing evaluation. Its relatively large size of program code section is probably the main reason for the low diffing rate. On the other hand, we report the average diffing rate of other cases is 43.87%, which is promising.

Table IV presents a summary of the performance data gathered from the validation process, showing the same set of metrics as we do pass selection in Section IV-C. While

TABLE IV: Mean of performance metrics for C programs in SPEC2006.

	bzip2	mcf	sjeng	libquantum	milc	lbm	sphinx3	perlbench	gcc	gobmk	hmmer	h264ref	**Mean**
Slowdown (%)	7.16	7.50	52.52	9.05	12.13	-2.30	2.61	75.48	107.90	37.70	2.82	10.73	26.94
Matching (%)	32.66	24.88	1.60	10.49	11.63	23.58	5.89	13.73	6.56	7.09	9.85	6.18	12.84
Pair.Match. (%)	44.89	33.51	37.80	52.70	43.01	31.88	38.34	88.82	38.98	45.37	51.87	60.73	47.32
Elim. (%)	99.05	96.42	100.00	99.21	99.64	97.26	99.71	100.00	99.98	100.00	99.82	100.00	99.26
Pair.Elim. (%)	96.47	90.69	98.27	97.12	98.46	92.46	98.93	99.76	99.90	99.80	98.99	99.27	97.51

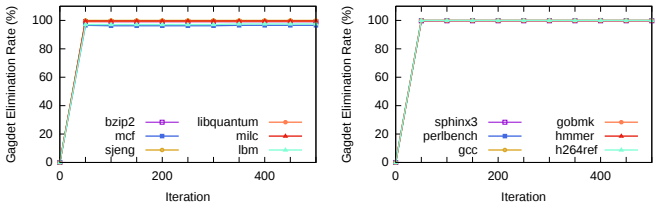


Fig. 16: ROP gadget elimination for SPEC binaries.

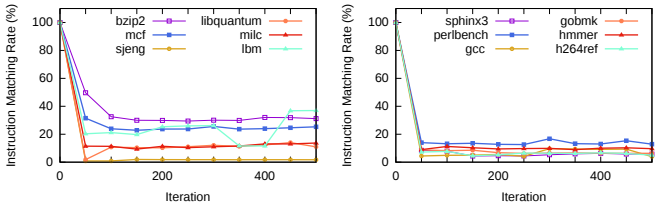


Fig. 17: Binary diffing for SPEC binaries.

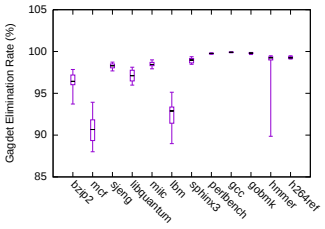


Fig. 18: Pairwise ROP gadget elimination for SPEC binaries.

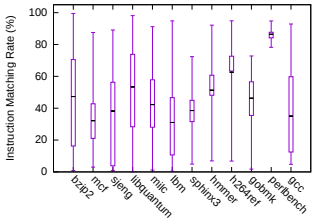


Fig. 19: Pairwise binary diffing for SPEC binaries.

on most metrics, the validation result is consistent with the selection process, there may be some concern about execution slowdown. For 8 out of 12 programs, composite diversification introduces less than 15% runtime overhead on average. However, the impact on the other 4 programs is much more significant, leading to an average slowdown from 37.70% to 107.90%. Our observation is that, programs with more complicated control flows tend to be penalized more by our diversification plan. This is under intuition since many passes in mix2 focus on disturbing the control flow.

Apart from metrics that have been used for pass selection, the processing time needed for generating diversified copies is also an important factor affecting the deployment of our framework. We report on average, it takes 248.1 seconds to process a binary in one iteration of diversification. The time is measured on the same machine to evaluate the runtime overhead, whose specification is posted in Section III-B. We also measure the relationship between the average processing time for one iteration and the original binary code size. We fit processing time with code size by a linear function with zero intercept. The regression test shows that the linear relation (with the slope as 0.695) is significant at the confidence level of 99%. Note that the processing time is the average of an *extreme* setting with 500 iterations. That means, as the average processing time increases almost linearly regarding the code size, we can expect a notable decrease in processing time when binaries are diversified with a smaller number iterations in the real-world usage.

VI. Discussion

Application Scope. A sample security application of software diversification is to mitigate the ROP attack. A number of defense techniques towards the ROP attack share similar idea to diversify the programs being protected to thwart attackers from easily guessing the addresses of ROP gadgets [28], [18], [11], [10], [17], [29]. However, some recent work has demonstrated that by leveraging memory disclosure vulnerabilities or even injecting gadgets through JIT compilers, it is still possible to construct ROP attacks on-the-fly [20], [21]. In other words, this so called Just-In-Time ROP (JIT-ROP) attacks can undermine the strength of existing static software diversification techniques.

Some recent works propose to harden the process runtime environment in order to defeat JIT-ROP attacks [30], [31], [32], [33], [34], [21]. Some of these mechanisms still require the software to be *diversified* before running. Thus, we can expect to boost existing advanced defenses through our proposed technique.

On the other hand, although in this paper we use the ROP gadget elimination rates to quantitatively evaluate our technique, software diversification indeed plays a critical role in many other program hardening techniques to defeat code tampering, memory corruption attacks, JIT compiler attacks, and reverse engineering [5]. Existing techniques in these categories can be complemented and enhanced by composite diversification.

Iterative Transformation. Our study shows that not all the variants can become more diversified when they are reprocessed for iterations. On the other hand, no significant cost increase is observed with more iterations. In our current implementation, we randomly select transformable candidates to process, and one lesson we learned is that composite transformation can consider taking the *already transformed program units* in priority, e.g., flattening a function's CFG which has already been flattened. Moreover, we consider the best practice to select transforming targets should be the combination of *random* selection from all the transformable candidates and *intentional* selection from the already transformed candidates.

There exist some diversifying methods which have finite transformable candidates, and the similarity rate would not de-

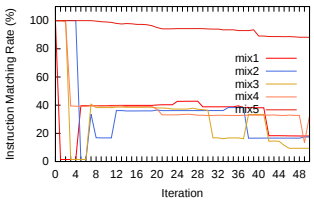


Fig. 20: Diffing results for 1-50 binaries.

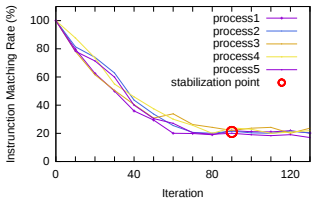


Fig. 21: Usage scenario: collect variants at the stabilization point.

crease after consuming all the candidates. A feasible improvement is to set a threshold of similarity decreasing rate in one iteration; if the similarity decreasing become insignificant, we can stop the iterations. Some kinds of transformations increase the execution cost with more iterations, and we refer readers to an orthogonal research, profile-guided diversification [35], to reduce the overhead. Besides, we can decrease the performance penalty by composing certain in-place diversifying transformations which have negligible cost [11]. We encourage software developers to adopt these optimization methods when deploying composite diversification frameworks in real-world scenarios.

Composition of Transformations. Many transformations can generate transformable targets for other methods, and by composing multiple methods, our experiments show that the performance of diversification is boosted noticeably. Also, even though we rule out three transformations at the first-stage selection because they do not perform well after a large number of iterations (§IV-B), iterating them for a large number of times might not be the only intended way of usage. For example, they can be applied only once at the end of each iteration to harvest their power in increasing diversity without incurring a high overhead. We encourage practitioners to leverage heuristics like this to refine the design of diversification frameworks.

Convergence of Iterative Transformations. Our pass selection has shown that composing different transformations leads to a synergy effect in composite diversification. One may be interested to know the convergence rate of the iteration, i.e., how many iterations are required to stabilize the diversity of generated variants. To preliminarily investigate the problem, we measure the binary diffing scores between the original `bzip2` and the first 50 variants produced by mix1 to mix5. The results are displayed in Figure 20. By bridging the data in Figure 20 and Figure 7, we find that the diversity stabilizes after about 40 iterations.

Finding the stabilization point can be helpful to improve the deployment strategy of composite diversification. According to our validation data, in some cases the cost of composite diversification has a non-negligible trend of growth as the rounds of iterations increase. For applications that are sensitive to such cost, the overhead may become unacceptable after a certain number of iterations. As shown in Figure 21, a possible solution is that users can start over the iteration from the original binary, and adopt the new variants generated after the point of stabilization. Since composite diversification randomly picks a transformation in each round, one can expect that variants produced after starting over would be different, whereas the cost will be controlled because of fewer iterations.

VII. Related Work

The idea of software diversification has been studied for decades. Diversifying approaches are presented with various scopes from a single instruction to the whole program. Fine-grained approaches such as instruction and basic block-level diversification aim at diversifying instructions within one basic block or sequences of basic blocks. Typical transformations include dead code insertion, instruction substitution, and basic block reordering [6], [7], [5]. These transformations have been adopted by both malware triage evasion [36], [37] and program randomization [11], [10]. Coarse-grained approaches are essentially deployed in the program runtime environment, hardening the program context from being exploited. Stack-layout randomization [9] and address space layout randomization (ASLR) [38] can deploy probabilistic defense, say, the unpredictable memory addresses can effectively impede code reuse attacks. However, attacks are still feasible due to limited randomization space of these coarse-grained approaches [39].

Other related work propose to randomize the encoding of program instructions [40]. Program encoding leverages one reversible encoding method to statically translate program text into encoded data. Usually a decoding routine and the decoding key is distributed inside the program, which can decode the data later. The encoded program can defeat a large number of static analyses. Improved techniques like virtual machine packer have been utilized to construct more secure binaries [41], [42], [43], [44].

Even though a few research work have touched the process of "composite diversification" [6], [22], [13], they are limited to some primary ideas or the proposed approaches only transform input program with very limited iterations. To our best knowledge, there are not existing work undertake an in-depth study on the composite transformation synergy.

VIII. Conclusion

Software diversification produces different variants of a program, which can effectively defeat code reuse attack and patch-based exploit generation. In this work, we initiate a new focus on this area, i.e., composite software diversification. Our in-depth study shows that composite diversification can outperform single-pass diversification in terms of better performance. We believe our study can provide useful guidelines for practitioners to design diversification tools in the future.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) under grant CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-13-1-0175, N00014-16-1-2265, and N00014-16-1-2912.

REFERENCES

[1] S. Designer, "Getting around non-executable stack (and fix)." 1997. [Online]. Available: http://seclists.org/bugtraq/1997/Aug/63

[2] R. Wojtczuk, "The advanced return-into-lib (C) exploits: PaX case study," *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*.

[3] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07, 2007, pp. 552–561.

[4] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. SP '08, 2008, pp. 143–157.

[5] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14, 2014, pp. 276–291.

[6] F. B. Cohen, "Operating system protection through program evolution," *Comput. Secur.*, vol. 12, no. 6, pp. 565–584, Oct. 1993.

[7] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, ser. HOTOS '97, 1997, pp. 67–75.

[8] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03, 2003, pp. 290–299.

[9] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security '12, 2012, pp. 40–50.

[10] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, 2012, pp. 157–168.

[11] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12, 2012, pp. 601–615.

[12] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August, "A framework for unrestricted whole-program optimization," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06, 2006, pp. 61–71.

[13] C. Collberg, S. Martin, J. Myers, and J. Nagra, "Distributed application tamper detection via continuous software updates," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12, 2012, pp. 319–328.

[14] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. USENIX Security '15, 2015, pp. 627–642.

[15] ——, "Uroboros: Instrumenting stripped binaries with static reassembling," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER '16, 2016, pp. 236–247.

[16] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to risc," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08, 2008, pp. 27–38.

[17] C. Kil, J. Jim, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, Dec 2006, pp. 339–348.

[18] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '13. ACM, 2013, pp. 299–310.

[19] "ROPgadget," http://shell-storm.org/project/ROPgadget.

[20] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-In-Time Code Reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. IEEE Computer Society, 2013, pp. 574–588.

[21] G. Maisuradze, M. Backes, and C. Rossow, "What cannot be read, cannot be leveraged? revisiting assumptions of JIT-ROP defenses," in *Proceedings of 2016 USENIX Conference on Security Symposium*, ser. USENIX Security '16, 2016, pp. 139–156.

[22] B. Coppens, B. De Sutter, and J. Maebe, "Feedback-driven binary code diversification," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 24:1–24:26, Jan. 2013.

[23] "BinDiff," http://www.zynamics.com/bindiff.html.

[24] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," in *Symposium sur la securite des Technologies de línformation et des Communications*, ser. SSTIC '05, 2005.

[25] H. Flake, "Structural comparison of executable objects," in *Proceedings of the IEEE Conference on Detection of Intrusions, Malware, and Vulnerability Assessment*, ser. DIMVA '05, 2005.

[26] E. W. Steyerberg, M. J. Eijkemans, and J. F. Habbema, "Stepwise selection in small data sets: A simulation study of bias in logistic regression analysis," *Journal of Clinical Epidemiology*, vol. 52, no. 10, pp. 935 – 942, 1999.

[27] J. M. Wagner and D. G. Shimshak, "Stepwise selection of variables in data envelopment analysis: Procedures and managerial perspectives," *European Journal of Operational Research*, vol. 180, no. 1, pp. 57 – 67, 2007.

[28] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd My Gadgets Go?" in *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2012, pp. 571–585.

[29] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proceedings of the 14th Conference on USENIX Security Symposium*, 2005.

[30] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 433–447.

[31] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z.Snow, and F. Monrose, "Isomeron: Code randomization resilient to (Just-In-Time) Return-Oriented Programming," in *22nd Annual Network & Distributed System Security Symposium (NDSS)*, 2015.

[32] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. ACM, 2014, pp. 1342–1353.

[33] K. Lu, S. Nürnberger, M. Backes, and W. Lee, "How to make ASLR win the clone wars: Runtime re-randomization," in *Proceedings of Symposium on Network and Distributed System Security*, ser. NDSS' 16, 2016.

[34] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Proceedings of 2015 IEEE Symposium on Security and Privacy (S&P)*, ser. IEEE S&P '15, 2015, pp. 763–780.

[35] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '13, 2013, pp. 1–11.

[36] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey," *CS701 Construction of Compilers*, 2005.

[37] E. Konstantinou and S. Wolthusen, "Metamorphic virus: Analysis and detection," *Technical Report RHUL-MA-2008-02*, vol. 15, 2008.

[38] P. Team, "PaX address space layout randomization (ASLR)."

[39] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04, 2004, pp. 298–307.

[40] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM conference on Computer and Communications Security*. ACM, 2003, pp. 272–280.

[41] O. Technologies, "Code Virtualizer," http://goo.gl/JLreyQ, 2003.

[42] VMPSofe, "Vmprotect," http://goo.gl/vXlYgy, 2004.

[43] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Proceedings of 2010 IEEE Symposium on Security and Privacy (S&P)*, 2009.

[44] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: a semantics-based approach," in *Proceedings of the 18th ACM conference on Computer and Communications Security*. ACM, 2011, pp. 275–284.