Principled Composition of Function Variants for Dynamic Software Diversity and Program Protection

Giacomo Priamo
priamo.1701568@studenti.uniroma1.it
Sapienza University of Rome
Italy

Daniele Cono D'Elia
delia@diag.uniroma1.it
Sapienza University of Rome
Italy

Leonardo Querzoni
querzoni@diag.uniroma1.it
Sapienza University of Rome
Italy

ABSTRACT

Artificial diversification of a software program can be a versatile tool in a wide range of software engineering and security scenarios. For example, randomizing implementation aspects can increase the costs for attackers as it prevents them from benefiting of precise knowledge of their target. A promising angle for diversification can be having two runs of a program on the same input yield inherently diverse instruction traces. Inspired by on-stack replacement designs for managed runtimes, in this paper we study how to transform a C program to realize continuous transfers of control and program state among function variants as they run. We discuss the technical challenges toward such goal and propose effective compiler techniques for it that enable the re-use of existing techniques for static diversification with no modifications. We implement our approach in LLVM and evaluate it on both synthetic and real-world subjects.

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Software and its engineering →** *Compilers*.

KEYWORDS

Software diversity, hardening, obfuscation, on-stack replacement.

ACM Reference Format:
Giacomo Priamo, Daniele Cono D'Elia, and Leonardo Querzoni. 2022. Principled Composition of Function Variants for Dynamic Software Diversity and Program Protection. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3551349.3559553

1 INTRODUCTION

Software diversity is a research field broadly investigated by different communities. At its essence, diversity can be regarded as the key to obtain properties such as resilience, performance, or novelty in heterogeneous contexts such as fault tolerance, software testing, performance tuning, and security [5, 15]. While natural diversity may spontaneously emerge in a software development ecosystem [5], automated diversity techniques can create and exploit diversity in existing code bases for different goals and scales.

Recent years have seen extensive applications of automated diversity at different software-stack levels, tackling problems involving code reuse attacks, information leaks, memory corruptions, and reverse engineering, among others [15]. According to their effects, automated techniques for diversification can be termed *static* when alternative versions of a program are produced and *dynamic* when a program is kept in a single version but its executions are diverse. Instruction substitution and stack frame layout randomization are two examples of static diversification techniques, whereas address space layout randomization and randomized heap allocators are examples of dynamic diversification techniques.

The attentive reader may have noticed that the static examples above alter the program structure, while the dynamic ones act on the runtime to obtain execution diversity indirectly. With the exception of just-in-time compilation scenarios, techniques that diversify *program instructions* at execution time can be difficult to design due to compatibility, complexity, performance, and other practical factors, especially if only the binary code can be distributed.

In this paper, we pursue a general-purpose design for dynamic diversity schemes for programs written in an imperative language. For a function hosting one or more regions that may benefit from diversification, we generate multiple function clones and make the corresponding regions in them be preceded by a query to an oracle, which can conditionally transfer control and program state to any clone. We then apply different static diversification transformations to each clone. In this design, multiple executions over the same input can bring highly diverse execution traces according to the runtime choices of the oracle and the compile-time transformations over the clones. Being general-purpose, this design may then be tailored to the specific needs of scenarios that naturally benefit from diversity, such as code obfuscation, information leakage mitigation, multi-variant execution systems, and tamper prevention [15].

Unfortunately, when it comes to real-world code, the potential benefits of such a "radical" design are naturally at odds with compatibility, implementation complexity, and performance aspects. For instance, two variants may substantially differ in the layout and values of their variables, requiring special-purpose compensation code to support state transfers between them. We note that similar issues are extensively studied in the Programming Language (PL) community for dynamic compilers, so as to support execution transfers between code variants compiled with different optimizations.

By revisiting PL concepts like *on-stack replacement* [8], we show how to realize the design above in a principled way, resulting in high compatibility with existing static transformations, limited implementation complexity, and satisfying performance. We suggest that one way to make run-time composition of function variants practical is to preserve the live variables and the memory storage of a program. To this end, we insert function calls acting simultaneously as barriers that preserve functional equivalence during transformations and as a means to transfer control and program state. We then show that by promoting stack variables to global memory, we can do away with explicit program state transfer, turning execution transfers into simple and efficient tail calls[1].

Contributions. This paper claims the following contributions:

- a principled approach to run-time composition of function variants for general software diversity and protection tasks;
- the identification of a careful combination of compiler techniques to ease its implementation in a mainstream compiler;
- a preliminary evaluation on compiler testing [28] and benchmarking [10] programs and on 5 real-world crypto subjects.

2 BACKGROUND AND RELATED WORKS

Experiments with software diversity date back the '70s for fault tolerance, followed eventually by a surge of interest for security applications since the '90s [5]. Several surveys attempt to cover the many facets of this body of works, e.g., recently [5, 15, 23]. The place (e.g., instructions, loops, functions, system) and the software life-cycle point (e.g., implementation, compilation, installation, execution) where artificial diversity is introduced can vary significantly among techniques [15]. This paper focuses on code diversification transformations applicable up to a function-level granularity.

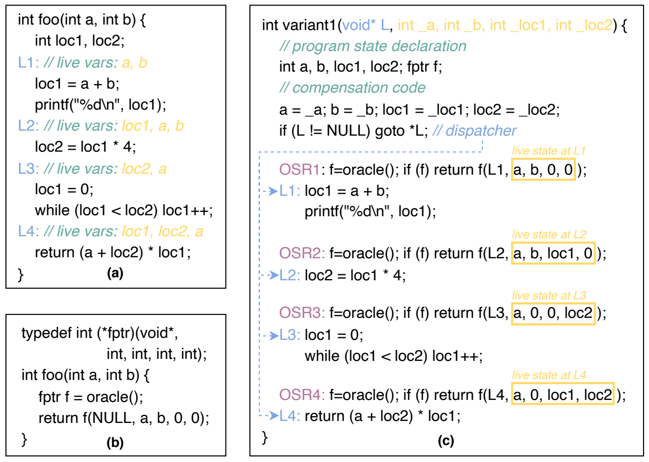
Examples of static diversifications include instruction reordering and substitution, garbage code insertion, opaque predicate insertion, function parameter randomization, and randomization of data at different levels (e.g., structure layout, heap layout) [3, 15]. Some literature considers as such also transformations of the likes of control flow flattening, which are usually studied as program protection techniques. Obfuscation shares many analogies with diversity, with many used transformations being the same [15], and is sometimes seen as different to diversity research because obfuscated program instances do not have to be kept private from adversaries [15].

As mentioned in Section 1, dynamic diversifications for binary instructions are presently scarce. We are aware of special-purpose embodiments for varying bytecode scheduling in virtualization obfuscation [13] and of in-place randomizations for crypto code [6]. The goal and contributions of this paper are different, as we study a general solution to accommodate arbitrary diversification techniques and support their composition in a sound and simple way.

Our approach produces programs where diversification techniques are stacked at run-time in arrangements every time different. Some of the challenges toward this goal have been studied in PL research. *On-stack replacement* (OSR) [8] techniques allow runtimes for managed languages (e.g., JVMs) to adaptively replace a function while it executes with a more (or less) optimized variant of it. Such runtimes normally enact OSR at favorable program points (e.g., loop back-edges) where state realignment is simpler and rely on glue code, stack frame replacement/extension, and other implementation devices to support execution transfers [7]. Loose analogies with our approach may also be recognized for research on dynamic software updating [11] and product programs [4].

3 APPROACH

This section details our principled approach for supporting the



```
int foo(int a, int b) {
    int loc1, loc2;
L1: // live vars: a, b
    loc1 = a + b;
    printf("%d\n", loc1);
L2: // live vars: loc1, a, b
    loc2 = loc1 * 4;
L3: // live vars: loc2, a
    loc1 = 0;
    while (loc1 < loc2) loc1++;
L4: // live vars: loc1, loc2, a
    return (a + loc2) * loc1;
}
```

```
typedef int (*fptr)(void*,
                    int, int, int, int);
int foo(int a, int b) {
    fptr f = oracle();
    return f(NULL, a, b, 0, 0);
}
```

```
int variant1(void* L, int _a, int _b, int _loc1, int _loc2) {
    // program state declaration
    int a, b, loc1, loc2; fptr f;
    // compensation code
    a = _a; b = _b; loc1 = _loc1; loc2 = _loc2;
    if (L != NULL) goto *L; // dispatcher

    OSR1: f=oracle(); if (f) return f(L1, a, b, 0, 0);
--> L1: loc1 = a + b;
        printf("%d\n", loc1);

    OSR2: f=oracle(); if (f) return f(L2, a, b, loc1, 0);
--> L2: loc2 = loc1 * 4;

    OSR3: f=oracle(); if (f) return f(L3, a, 0, 0, loc2);
--> L3: loc1 = 0;
        while (loc1 < loc2) loc1++;

    OSR4: f=oracle(); if (f) return f(L4, a, 0, loc1, loc2);
--> L4: return (a + loc2) * loc1;
}
```

Figure 1: Running example. The figure parts are: (a) original target function `foo`, (b) modification of `foo` for call redirection, (c) template for dynamic variant composition over `foo`.

general-purpose design for dynamic diversity anticipated in Section 1. After discussing the main technical challenges behind the design, we illustrate a careful combination of compiler techniques that can significantly ease the realization of concrete embodiments of the design. We use the code in Figure 1 as running example.

Function `foo` from Figure 1(a) comes with 4 code regions delimited by annotations in the form of C labels. Let us suppose we are interested in dynamically diversifying each such region: ideally, every time the control flow reaches an annotated region entry point, we seek a different variant of a region to be selected and executed. The choice of which variant to use at a given moment is delegated to an **oracle**. The *granularity* for diversification is fine-grained: annotations can delimit arbitrary portions of basic blocks and compositions thereof (e.g., the region starting at L3 includes a basic block with a single predecessor and a loop), but possibly also leave out regions that would not benefit from diversity.

Challenges. A critical choice in the design is what scope one may allow in diversification transformations. On one end, simple embodiments can apply special-purpose local transformations to individual regions, so that no side-effect escapes such scope (i.e., the instructions and data of other regions are unaffected) and control transfers among regions is greatly simplified. However, there are obvious limitations in the extent of the diversity that such a choice may allow. For instance, in code obfuscation, an adversary may quickly learn from multiple executions that the sequences are interchangeable and thwart their diversity with syntactic replacements.

On the opposite end of the spectrum, complex embodiments could support heavy-duty diversifications that alter the control and/or data dependencies among regions, for instance by hoisting or sinking portions of computations to other regions and by merging whole regions. While this choice could bring obvious benefits in the dynamic diversity of execution traces, supporting program state realignment between deeply diverse variants is a daunting prospect, since the effects of each diversification should be tracked, understood, and possibly compensated for, across control flow transfers.

[1]We make available a prototype in LLVM at: https://github.com/gpriamo/osr-diversity.

Call Barriers and Live Variables. In this paper, we show that variants obtained with generic diversification transformations can be composed in a principled way without requiring a runtime component that logs program state or assists with transfers. To this end, we revisit two key concepts from the dynamic compilers literature.

The first concept are *live variables.* D'Elia and Demetrescu proved in [8] that reasoning only about the live variables that two semantically equivalent functions have in common at a program point can suffice to allow for transferring execution between them at it.

The second concept are *barriers.* Pioneered in [12] for debugging optimized code in SELF, a barrier is a construct that forces a dynamic compiler to track and (optionally) preserve program state values at specific places. As an example, the Jikes RVM uses barriers when lowering the bytecode for an interruptible method, for instance to hold live variables before an inlined call [9, 20]. Barriers are not strictly limiting for the work of an optimizer, which is allowed to make semantics-preserving code changes also across barriers.

For imperative languages, we can effectively combine the two concepts as shown in Figure 1(c). We first identify the variables that are live when entering each region to diversify. We create a **variant template** as a clone of the target function and modify it as follows:

- we edit its prototype to receive a destination label and a value for every variable in the superset of live variables of interest;
- before the start of each region, we add **OSR points** to conditionally transfer execution to a variant chosen by the oracle, an operation done by calling a function that receives the label of such region and the current values for the variables live at it (using, e.g., a zero value for the remaining variables);
- we edit its entry block such that it assigns the incoming values to the corresponding live-variable storage and, when the incoming destination label is valid, it jumps to it.

Finally, we modify the original function to ask the oracle for a variant instance and transfer control to it, passing as arguments its own call parameters and a void destination block—see Figure 1(b).

Properties. The advantages of our approach are several. First, it comes with high compatibility properties: existing static diversification techniques can be used off the shelf on each variant and, as we detail next, it can see platform-independent embodiments in the intermediate representation of a compiler. Moreover, the approach is arguably simple, as it does not require a runtime component to handle program state but only leverages the semantics of imperative languages like C. After compilation, each variant can even be further manipulated with binary-level methods, including complex ones such as commercial-strength virtualization obfuscation [24].

The correctness of the approach follows from using function calls not only as a control transfer technique, but also as barriers: to preserve the *observable behavior* of the code, a semantics-preserving code transformation cannot alter function call argument values if the call target is uncertain (e.g., external). These effects are similar to using the C *volatile* keyword for global storage accesses to prevent heavy-duty code optimizations from altering visible values. Indeed, to preserve the memory model, we opt for conservatively marking as *volatile* all pointer values in a variant template's prototype.

Finally, the approach is flexible in several respects. The oracle can model policies for different deployment scenarios. In its simplest form, it can choose with a probability whether, and to which variant, to transfer execution. However, it may also dictate transitions between specific variants according to particular execution conditions. Our approach also enables subsequent edits on variants such as removing specific OSR points or even entire regions, producing partially functional variants. We continue this discussion in Section 5.

Making it Practical. We faced several challenges to implement our approach in a practical and efficient way. Without loss of generality, we discuss solutions in the context of the LLVM compiler and its intermediate representation (IR). For starters, using classic calls to transfer program state can be expensive when done frequently and lead to stack frame proliferation. Furthermore, live variables can substantially differ across multiple OSR points, leading to potentially many values to transfer during calls—yellow boxes in Figure 1(c). Those, in turn, would need heavy-duty IR manipulations upon their assignment—*compensation code* block in Figure 1(c)—to comply with the SSA form [21] in use to the variants.

To mitigate the burden of transferring live variables as call arguments, one could opt for spilling variables to global storage as in early LLVM OSR schemes [14]. However, this operation can still be costly (and conspicuous) if done frequently. We choose instead to promote all local variables to *volatile* global storage, effectively translating the function frame away from the stack. By doing so, the compiler can reuse the stack frame of the currently executing variant, emitting a tail call that becomes an indirect jump in binary code. Two main benefits follow: the program keeps the contents of such locations up to date as part of its normal operation and we can avoid bookkeeping costs for OSR stack frames altogether.

Implementation. In our prototype, we take as input the IR file generated at any optimization level for the C function(s) to diversify and a list of basic blocks for OSR point insertion. After creating a template variant by cloning the original code, we apply the `reg2mem` pass on it to expose its local variables as `alloca` instructions and promote them to global memory. Upon entering a basic block, there are no live path-dependent variables (i.e., ϕ*-nodes*) in the transformed code, hence the global storage suffices to support transitions between variants[2]. We now add OSR machinery at each basic block of interest to invoke an oracle and, if the oracle returns a code pointer, make a tail call to it followed by a `ret`, so as to chain the propagation of return values among each invoked variants.

These implementation choices reduce spatial and temporal management overheads for state keeping and execution transfers, making the implementation of our approach in a mainstream compiler practical. We wrote the current prototype using 830 C++ LOC.

As for the oracle, we populate at run-time a table of function pointers and at each OSR point we choose with a probability value p whether to make a transition, drawing uniformly at random from N available variants for each diversified function. For simplicity, we currently rely on the C library function `rand()` to model p.

Limitations. Our prototype uses fixed locations for stack variable promotion. This implies that we cannot diversify functions that may be used concurrently or be involved in direct or mutual recursion. A solution can be maintaining a per-thread stack of structures hosting instances of promoted variables and having each active instance of a diversified function access its stack entry to retrieve the storage.

[2]Alternatively, one could spill ϕ-nodes to global storage and use LLVM's `SSAUpdater` to make them reached by values from compensation code. We avoided this complexity.

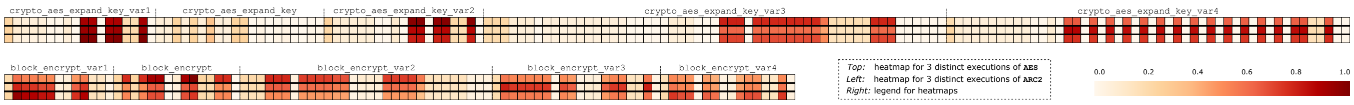


Figure 2: Heatmap of normalized execution frequencies for basic blocks of diversified variants of a core `aes` and `arc2` function.

4 PRELIMINARY EXPERIMENTAL ANALYSIS

We conduct three sets of experiments for: **a)** stressing our implementation by diversifying Csmith programs, **b)** measuring overheads on performance-critical code, and **c)** exploring dynamic diversity when applying our approach to real-world subjects. As code transformations, we draw from 12 LLVM intra and inter-procedural optimizations (e.g., loop rotation, loop unrolling, SCCP [19]) and 5 obfuscations (basic block splitting, bogus control flow insertion, function call wrapping, instruction substitution, variable substitution) that we extracted from LLVM-based obfuscators [1, 2]. For simplicity, we apply them to the whole code of variant instances.

a) We build an automated pipeline to insert OSR points at random locations in compiler stress-testing code and generate up to 8 variants for each diversified function. We choose different sets of transformations on each variant and vary the optimization settings when compiling the IR to binary code. We use 500 Csmith [28] programs as seeds for the pipeline and repeatedly tested them in different assortments. For a subset of them, we also verify that VMProtect (a heavy-duty, state-of-the-art [27] commercial obfuscator) manipulates our diversified compiled code correctly. No error occurred.

b) We consider a worst-case scenario where we apply our approach to performance-critical code. To isolate the impact of our machinery for variant composition, we generate 8 identical variants. We study the Computer Language Benchmark Game (`clbg`) [10] subjects that are available in a single-threaded C version. We ran the tests using OSR probability values $p \in \{0.1, 0.25, 0.5\}$ and two optimization levels on a Linux machine with an Intel i7-10875H CPU and negligible background activity. For OSR point insertion, we target basic blocks in loops with high trip counts and in frequently executed functions as fall-back. Table 1 reports the OSR points and rate and the average slowdown on 10 runs of each subject. We omit 95% confidence intervals as we observed insignificant deviations.

In short, the slowdown is hardly noticeable when OSR transitions are not excessively frequent (OSR_{rate} column): this is impacted by the probability p of making an execution transfer and predominantly by the program-specific characteristics of the regions where OSR points are placed. While follow-up research should study trade-offs between overheads and resulting diversity from OSR point selection, we find these performance numbers already promising.

By analyzing the executions with `perf`, we found that typically over half of the additional execution time is spent in the `rand()` function. This time may be reduced by, e.g., using an own PRNG [6] or inlining the oracle (we measured average saves of ~20%), or be used to accommodate more complex oracles, e.g., using *dynamic opaque predicates* [17] that see execution-dependent truth values.

c) We consider a recurrent code obfuscation scenario where the target resembles a license key validation sequence. We diversify 5 crypto programs used in other security literature [25, 26]: `aes` and `anubis` from the Chronos library, the `loki91` cipher reference, and `arc2` and `blowfish` from the pycrypto toolkit. We take a core function in them and diversify it using optimization and obfuscation transformations different across variants. We add a testing driver to realize 100 end-to-end crypto cycles and use DBI [16] to profile the executed basic blocks. We normalize the counters in a [0, 1] value first by using the highest count in the enclosing function and then globally by using the call count of the mostly invoked variant.

Table 1: Statistics on `clbg` programs. Modified functions are reported in "( )" in *Points*. OSR events are counted per μsec.

		OSR_{rate}	Slowdown (%) at `-O0`			Slowdown (%) at `-O1`		
Program	Points	p=0.5	p=0.1	p=0.25	p=0.5	p=0.1	p=0.25	p=0.5
bintrees	2 (1)	$\sim 10^{-6}$	-1.74	-3.49	-5.00	-2.33	-3.45	-3.59
fannkuch	4 (2)	$6 \cdot 10^{-6}$	3.26	2.32	2.34	9.54	10.42	9.65
fasta	4 (4)	10.56	10.44	18.77	37.03	66.15	79.42	106.99
mbrot	2 (2)	1.86	4.21	7.14	8.55	58.29	63.26	63.04
n-body	4 (3)	13.54	28.03	50.12	77.63	97.10	134.60	192.14
pidigits	3 (1)	$3 \cdot 10^{-2}$	1.47	3.43	2.65	-2.74	-2.83	0.76
regex	3 (3)	$8 \cdot 10^{-6}$	1.21	1.53	4.29	-1.49	-0.91	-2.02
revcomp	2 (1)	6.15	3.66	5.51	7.73	11.52	17.93	30.69

For space reasons, in Figure 2 we report heatmaps only for 3 runs of `aes` and `arc2` diversified using 4 variants and p=0.25. A heatmap visualizes how these runs covered each basic block (sorted by address) with varying frequencies. As for performance, crypto cycles completed in a bounded time of the same order of magnitude of the original code, suggesting that our scheme can be practical (and possibly even amenable for using heavy-duty static obfuscations, some of which can induce two-digit slowdowns [22]).

5 FUTURE PLANS

Alongside the above-mentioned trade-offs in OSR point insertion, a compelling question to investigate next involves the oracle logic. Different usage scenarios of our methods may demand different strategies to harness diversity. A probability-based model may work fairly well in certain information leakage scenarios or in multi-variant systems. In program protection scenarios where adversaries have full execution visibility (e.g., trace recording), the oracle could be strengthened in different ways. For instance, adding classic [3] and, better yet, dynamic opaque predicates or MBA expressions [3] to the oracle's logic may induce state explosion in state-of-the-art trace deobfuscation methods [27] that try to reconstruct a simplified program representation by merging multiple simplified traces.

More interestingly, an oracle may be programmed to check program state conditions and decree transitions only for specific combinations of variants and program locations. With adversaries unaware of such logic, one may then alter the semantics of specific regions in some variants, inducing bogus executions whenever adversaries try to force their exploration as in some attack methods [18].

Finally, to extend the methodology, one may explore new diversifications that alter observable storage values (or scramble its layout) and synthesize realignment code for when entering other variants.

Acknowledgements. This work has been partially supported by the IoT-STYLE project RG12117A7CE68848.

REFERENCES

[1] 2019. *Hikari Rebooted.* https://github.com/HikariRebooted/HikariCore.

[2] 2022. *Pluto Obfuscator.* https://github.com/bluesadi/Pluto-Obfuscator.

[3] Sebastian Banescu and Alexander Pretschner. 2018. Chapter Five - A Tutorial on Software Obfuscation. *Advances in Computers* 108 (2018), 283–353. https://doi.org/10.1016/bs.adcom.2017.09.004

[4] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *Proc. of the 17th International Conference on Formal Methods (FM'11).* Springer-Verlag, 200–214.

[5] Benoit Baudry and Martin Monperrus. 2015. The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond. *ACM Comput. Surv.* 48, 1, Article 16 (sep 2015), 26 pages. https://doi.org/10.1145/2807593

[6] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *22nd Annual Network and Distributed System Security Symposium, NDSS.* The Internet Society. https://www.ndss-symposium.org/ndss2015/thwarting-cache-side-channel-attacks-through-dynamic-software-diversity

[7] Daniele Cono D'Elia and Camil Demetrescu. 2016. Flexible On-stack Replacement in LLVM. In *Proc. of the 2016 International Symposium on Code Generation and Optimization (CGO '16).* Association for Computing Machinery, 250–260. https://doi.org/10.1145/2854038.2854061

[8] Daniele Cono D'Elia and Camil Demetrescu. 2018. On-Stack Replacement, Distilled. In *Proc. of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018).* Association for Computing Machinery, 166–180. https://doi.org/10.1145/3192366.3192396

[9] Stephen J. Fink and Feng Qian. 2003. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *Proc. of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '03).* IEEE Computer Society, 241–252.

[10] Isaac Gouy. 2018. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/ (accessed: May 27, 2022).

[11] Michael Hicks, Jonathan T. Moore, and Scott Nettles. 2001. Dynamic Software Updating. In *Proc. of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01).* Association for Computing Machinery, 13–23. https://doi.org/10.1145/378795.378798

[12] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proc. of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92).* Association for Computing Machinery, 32–43. https://doi.org/10.1145/143095.143114

[13] Kaiyuan Kuang, Zhanyong Tang, Xiaoqing Gong, Dingyi Fang, Xiaojiang Chen, and Zheng Wang. 2018. Enhance Virtual-Machine-Based Code Obfuscation Security through Dynamic Bytecode Scheduling. *Comput. Secur.* 74, C (may 2018), 202–220. https://doi.org/10.1016/j.cose.2018.01.008

[14] Nurudeen A. Lameed and Laurie J. Hendren. 2013. A Modular Approach to On-Stack Replacement in LLVM. In *Proc. of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '13).* Association for Computing Machinery, 143–154. https://doi.org/10.1145/2451512.2451541

[15] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *Proc. of the 2014 IEEE Symposium on Security and Privacy (SP '14).* IEEE Computer Society, 276–291. https://doi.org/10.1109/SP.2014.25

[16] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05).* Association for Computing Machinery, 190–200. https://doi.org/10.1145/1065010.1065034

[17] J. Palsberg, S. Krishnaswamy, Minseok Kwon, D. Ma, Qiuyun Shao, and Y. Zhang. 2000. Experience with Software Watermarking. In *Proc. of the 16th Annual Computer Security Applications Conference (ACSAC '00).* IEEE Computer Society, 308.

[18] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications. In *Proc. of the 23rd USENIX Conference on Security Symposium (SEC'14).* USENIX Association, 829–844.

[19] LLVM Project. 2022. *LLVM's Analysis and Transform Passes.* https://llvm.org/docs/Passes.html (accessed: May 27, 2022).

[20] Feng Qian. 2005. *Runtime Techniques and Inteprocedural Analysis in Java Virtual Machines.* Ph. D. Dissertation. McGill University.

[21] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proc. of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88).* Association for Computing Machinery, 12–27. https://doi.org/10.1145/73560.73562

[22] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. 2018. Symbolic Deobfuscation: From Virtualized Code Back to the Original. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc (Eds.). Springer International Publishing, 372–392.

[23] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software Diversity: State of the Art and Perspectives. *Int. J. Softw. Tools Technol. Transf.* 14, 5 (oct 2012), 477–495. https://doi.org/10.1007/s10009-012-0253-y

[24] VMProtect Software. 2017. *VMProtect.* https://vmpsoft.com/.

[25] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. 2018. DATA–Differential Address Trace Analysis: Finding Address-Based Side-Channels in Binaries. In *Proc. of the 27th USENIX Conference on Security Symposium (SEC'18).* USENIX Association, 603–620.

[26] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating Timing Side-Channel Leaks Using Program Repair. In *Proc. of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018).* Association for Computing Machinery, 15–26. https://doi.org/10.1145/3213846.3213851

[27] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proc. of the 2015 IEEE Symposium on Security and Privacy (SP '15).* IEEE Computer Society, 674–691. https://doi.org/10.1109/SP.2015.47

[28] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11).* Association for Computing Machinery, 283–294. https://doi.org/10.1145/1993498.1993532