LETTER

Generation of Efficient Obfuscated Code through Just-in-Time Compilation

Muhammad HATABA[†*a)], ***Member***, **Ahmed EL-MAHDY**[†,††], ***Nonmember***, ***and*** **Kazunori UEDA**[†††], ***Member***

SUMMARY Nowadays the computing technology is going through a major paradigm shift. Local processing platforms are being replaced by physically out of reach yet more powerful and scalable environments such as the cloud computing platforms. Previously, we introduced the OJIT system as a novel approach for obfuscating remotely executed programs, making them difficult for adversaries to reverse-engineer. The system exploited the JIT compilation technology to randomly and dynamically transform the code, making it constantly changing, thereby complicating the execution state. This work aims to propose the new design iOJIT, as an enhanced approach that patches the old systems shortcomings, and potentially provides more effective obfuscation. Here, we present an analytic study of the obfuscation techniques on the generated code and the cost of applying such transformations in terms of execution time and performance overhead. Based upon this profiling study, we implemented a new algorithm to choose which obfuscation techniques would be better chosen for "efficient" obfuscation according to our metrics, i.e., less prone to security attacks. Another goal was to study the system performance with different applications. Therefore, we applied our system on a cloud platform running different standard benchmarks from SPEC suite.

key words: *cloud computing security, dynamic compilation, obfuscation, optimization transformations, side-channels*

1. Motivation

With the emergence of new computing technologies, such as cloud computing, fog computing and Internet of things (IoT), we need to rethink the way we execute our programs. Often, these new platforms are not totally governed by the end user. Hence comes trustworthiness doubts and security concerns [1]. Relying solely on cryptographic key encryption is not enough because at some point of the processing time decryption could be done remotely, thereby opening all defenses for inside attackers. Thus we proposed the use of the security-by-obscurity paradigm, which depends on hiding system internals from attackers via some sort of obfuscation to confuse attackers.

2. Threat Model

Our main concern is the widely infamous side-channel attacks [4] which could be mounted to a SCARE attack [16] (Side-Channel Attacks Reverse-Engineering); a threat typically present in remote execution platforms such as the cloud computing environment.

Hardware Assumption: we assume that an attacker would be able to surreptitiously implant himself inside the computing environment either via intrusion mechanisms or the attacker himself being a legitimate user of the platform or even a cloud administrator. Then, he would try to access the shared physical resources and examine its usage patterns to learn information about the execution trace of the running programs. He could even examine physical properties of the system such as power consumption, electromagnetic radiation or even sound emissions to infer knowledge about the executed instructions.

Software Assumption: we assume that the attacker does not have access to the binary itself or prior knowledge about the victim program or how it was compiled, but he can run malicious applications to probe its runtime statistics hoping that the code behavior would be producible under the same circumstances and the same input data, then he may predict the execution trace of the program or reverse-engineer it.

3. Related Work

Most countermeasures to impede side-channel attacks fall into two main categories: (1) eliminate the information leakage itself by means of hardware shielding modifications or adding more access controls, e.g: [5]. (2) eliminate the correlation between the leaked information and the protected program. This could be done by designing the software to be isochronous, having a constant execution time regardless of input data, but clearly this is not practical. On the contrary, non-isochronous techniques try to introduce random noise to the code execution such as NOP or sleep instruction, but this is easily detectable in cache and power analysis attacks. Further compiler based techniques may involve variable latency instruction [8] or branch eliminations [9]. Other techniques investigated software diversity to thwart cache-side channels [10]. But most of these techniques focused on protecting secret data rather than the code itself. There are obfuscation techniques to impede disassembling software such as [12] and [11], but most of these techniques are applied statically and may not be satiable for a remote-setup such as the cloud.

On the other hand, our software-based system which

Manuscript received August 24, 2018.
Manuscript revised October 19, 2018.
Manuscript publicized November 22, 2018.

[†]The authors are with Egypt-Japan University of Science and Technology (E-JUST), Alexandria, Egypt.
[††]The author is with Alexandria University, Alexandria, Egypt.
[†††]The author is with Waseda University, Tokyo, 169–8555 Japan.
[*]Presently, also with National Telecommunication Institute (NTI), Cairo, Egypt.
a) E-mail: mohamed.hataba@ejust.edu.eg

DOI: 10.1587/transinf.2018EDL8180

provides defense against reverse engineering by means of dynamically yet randomly obfuscated code diversification, which entails probabilistic protection against analysis tools, tailored to input program characteristics in a simple yet effective technique, verified by software complexity measures.

4. OJIT Background

Code obfuscation was introduced quite a while ago [6]. It is widely popular among virus and malware developers to evade virus scanners and protection programs via hiding the true propose of the executed code. OJIT [2] used the same concepts to dynamically produce functionally equivalent versions of a running program, but they differ in their appearance and behavior, as per our metrics, making them very complicated for an attacker to understand or reverse-engineer.

We built our obfuscation system as an extension to the JIT compiler of the LLVM compilation framework [7], which both open-source and platform-independent. Moreover, LLVM has multiple front-end and back-end compiler capabilities that support a wide range of high-level programming languages such as the C family and an even wider range of hardware architectures. That's why we choose for our system to work primarily on the LLVM IR code level. All of these characteristics makes our system generic and quite suitable for remote execution platforms such as the cloud computing.

We modified the Execution Engine of LLVM, forcing it to call the obfuscated JIT compiler (OJIT) every time a compilation unit (typically a module or a function) is invoked, such that every time we JIT a function/module, we effectively obfuscate it. Thus, we utilize the jitting time in adding more distortion to the observable total running time. Most of these obfuscation techniques are based on the standard LLVM compiler optimizations called transformation passes. We also developed more innovative techniques in [3].

For simplicity back then, we developed an algorithm to select a random set of a random size of obfuscation passes to be applied in a random order to the input program. In most cases, the resulting obfuscated code versions were quite different from the original code and from each other as well. This type of code diversification was measured in terms of software complexity metrics such as instruction count, branch count and decision points along with the more complicated similarity metric measured by Stanford's (MOSS) system [14]. In addition, we predicted that this diversification would typically result in execution time changes between different code versions as a result of code behavior being changed, so we measured that as well.

Nevertheless, the OJIT system was adequate but not optimal; that is, in some occasions the resulting obfuscated code was not that very different from the original one because the system did not choose the most effective obfuscation transformations according to our metrics, since the selection process was merely random. For example, the OJIT system would choose loop related transformation for a program which had no loops or recursive function transformations when none existed. That is why we needed a more efficient way to examine the input code and refine the set of selectable transformations that would produce a greater change in the obfuscated versions as per the metrics we discussed earlier.

Algorithm 1 Pseudocode of iOJIT Algorithm

```
1: INPUT: Profiling Data (T), Threshold , Length
2: OUTPUT: Obfuscation Sequence P_SL
3: P_i ← 0, P_s ← 0, k ← 0, P_B ← 0
4: while k < Length do
5:     P_i ← P_B, Candidates ← ∅, T_B ← T(P_i, P_s)
6:     for all P_j ∈ Passes do
7:         T_N ← T(P_s, P_j), ΔT ← ((T_N − T_B)/T_B) × 100
8:         if |ΔT| ≥ Threshold then
9:             Candidates ← Candidates ∪ P_j
10:    P_B ← P_s
11:    if Candidates = ∅ then
12:        P_s ← GetMax(P_i, P_s, T)
13:    else
14:        P_s ← Toss(Candidates)
15:    k ← k + 1
16:    P_SL ← P_SL ∪ P_s
17: end
```

5. Our New Enhanced System: The iOJIT

Our purpose here is to enhance the Obfuscated JIT compiler so that it would dynamically produce efficiently obfuscated code versions. Since our major threat (side-channel attacks) relied mainly on statistical timing analysis of the running code, we selected the total time as our primary metric for the effectiveness of the obfuscation transformations. Consequently, our modified system introduces a profiling step, where we applied all possible pairs of obfuscation transformations in separate code runs to study their effect on the input program's execution time. This helps us to determine the candidate effective obfuscation transformations and their suitable order.

Figure 1 shows an overview of our proposed system's compilation steps. The front-end compiler takes an input program consisting of a number of code units in its high-level programming language form and compiles it into the

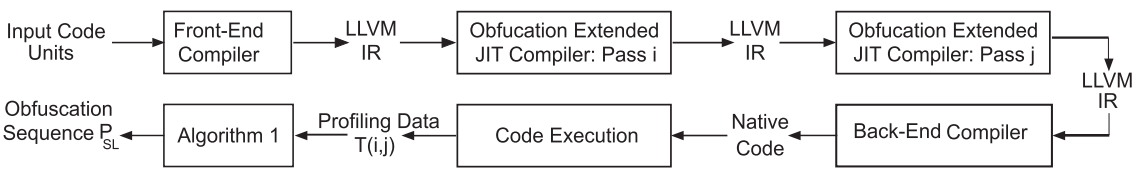


Fig. 1 iOJIT compilation steps.

LLVM Intermediate Representation (IR). This is fed into our obfuscation extended JIT compiler, which then applies a pair of transformation passes (Pass i and Pass j) and produces an obfuscated code version also in the LLVM IR. Finally, the back-end compiler generates the native code version which is ready for execution. This process will be repeated until all units of a the given program are executed and all possible transformation pairs are tested, and every time the total execution time is measured. Hence, we have all the profiling data needed for the following phases of our system.

In addition, we developed the algorithm shown above (Algorithm 1), which studies the profiling data to choose an efficient sequence of obfuscation transformations, which results in more disruptions to the execution time of the code. This effect is measured as a normalized percentage of the change in execution time $\Delta T\%$ after applying the transformations as shown in Eq. (1), where T_N is the execution time of the code version after applying the obfuscation transformation, and T_B is the execution time of the code version before that.

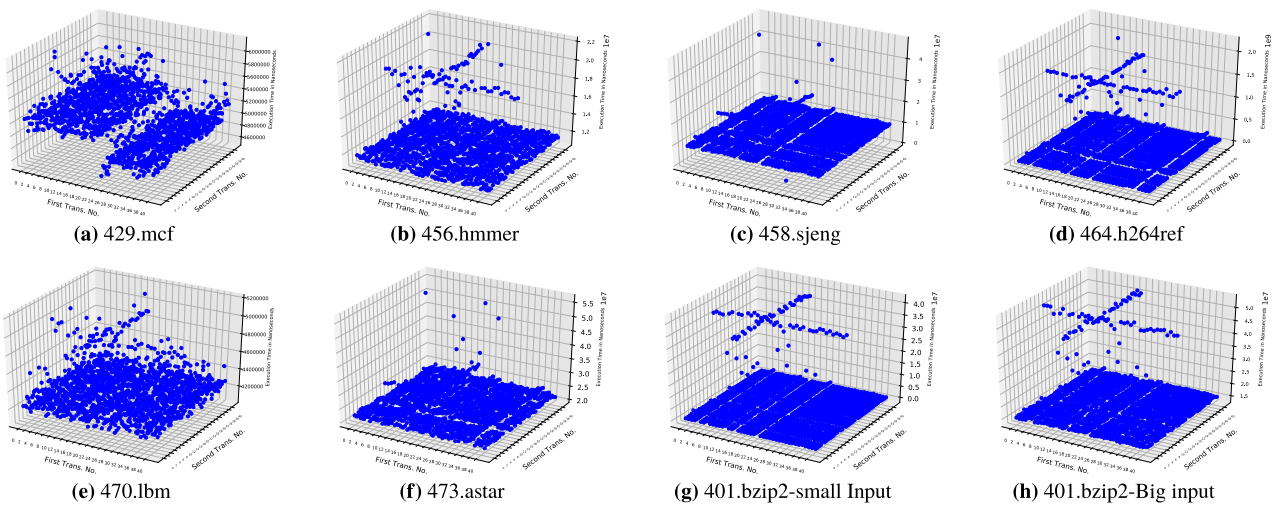
$$\Delta T\% = ((T_N - T_B)/T_B) \times 100 \quad (1)$$

The algorithm starts from the pair (0,0), which corresponds to the original code version where 0 means no transformation is applied. Then we search all other pass pairs (0, P_j), where P_j is the next pass number, to find out the pool of candidate passes, which if applied, would have a tangible $\Delta T\%$ (i.e. exceeding the predefined *Threshold* value). Then the function **Toss** takes this set of candidate passes and returns a randomly selected one of these passes P_s. If none exists, we select the pass which had the largest $\Delta T\%$ using the function **GetMax**. Then, we repeat the selection process starting from the newly introduced obfuscation pair (P_s, P_j) until we reach the arbitrarily predefined *Length* of the efficient obfuscation sequence P_{SL}. Hence, we can apply subsets of these transformations in that order expecting that each execution version of the code would have quite different execution time.

6. Experimental Results and Analysis

We applied our system to an arbitrary set of programs selected from the standard SPEC CPU 2006 benchmarks suite [15]. Most of these programs are integer benchmarks, but a few are float. Some programs are written in C++, other in plain C. The reason behind such miscellaneous selection is to allow for a diversity of code characteristic, thus making our system as generic as possible. For the sake of simplicity, we used a small input data from the standard test data set provided by SPEC. The platform we are working with is a Virtual Machine having 16 cores and 16 GB RAM, operated by CentOS 7.2, and hosted by an OpenNebula 5.4.1 cloud environment.

Here we set an arbitrary length for the obfuscation sequence of 10 transformations. This can be extended or shortened as per our obfuscation needs. From the profiling step, we obtained 3D scatter graphs illustrated in Fig. 2. These represent the execution timing after applying a set of 41 by 41 passes. We noticed that many of these passes are likely to be "idempotent", i.e., applying the same pass twice would not change the code. Aside from that, there was quite a change in execution times, depending on the input programs characteristics. The results are depicted in Table 1, which shows the selected transformation numbers and their corresponding normalized timing effect in percentage after applying it. We can see that in some benchmarks the change $\Delta T\%$ is quite large, such as in the 464.h264ref benchmark, reaching up to 92% the previous execution time (as a slowdown), in other cases there would be a speed up of nearly −57% for that same program, with the same input data, which is a drastic change. However, if this much slowdown is undesirable, it could be mitigated by providing a predefined ceiling (or in other words, a cost or a performance overhead) that a user would be willing to endure, which would control the selection of the effective sequence.



(a) 429.mcf **(b)** 456.hmmer **(c)** 458.sjeng **(d)** 464.h264ref

(e) 470.lbm **(f)** 473.astar **(g)** 401.bzip2-small Input **(h)** 401.bzip2-Big input

Fig. 2 Execution times of obfuscated versions due to consecutive transformation pairs.

Table 1 Effective obfuscation sequences for each benchmark and their timing effect.

429.mcf	P_S	25	24	18	12	24	16	20	34	16	9
	$\Delta T\%$	−10.3	−11.98	10.22	11.04	−10.24	10.51	13.53	−12.62	14.62	15.06
456.hmmer	P_S	16	22	27	23	32	7	16	29	26	16
	$\Delta T\%$	64.48	−10.6	−33.76	10.58	79.45	−44.11	46.2	−13.79	−29.66	72.12
458.sjeng	P_S	23	4	1	2	4	11	5	31	16	7
	$\Delta T\%$	22.55	18.4	−15.9	−20.04	22.15	12.45	−15.54	−12.41	73.16	−10.9
464.h264ref	P_S	13	4	18	2	34	12	7	2	3	14
	$\Delta T\%$	92.02	67.5	−25.79	−57.28	54.79	10.54	−42.14	−42.65	−11.43	10.31
470.lbm	P_S	16	4	2	16	10	19	16	38	32	33
	$\Delta T\%$	14.88	−5.3	−5.94	15.61	−7.97	−8.26	19.07	−5.36	−13.12	−14.81
473.astar	P_S	16	32	11	36	12	11	9	24	11	4
	$\Delta T\%$	22.59	11.88	−16.42	15.62	−18.13	19.31	−15.37	−11.47	11.24	10.15
401.bzip2-s	P_S	6	7	4	12	25	32	22	28	17	11
	$\Delta T\%$	15.72	47.43	132.4	10.33	−66.22	−31.25	54.21	−14.53	−17.77	49.8
401.bzip2-B	P_S	16	23	39	24	4	16	5	39	16	23
	$\Delta T\%$	223.91	−30.8	−53.86	−10.66	10.94	124.13	44.32	−70	235.91	−29.94

Table 2 $\Delta T\%$ statistics for the 401.bzip2 benchmark over the course of 50 runs.

Code Version	$V_{16,23}$	$V_{23,39}$	$V_{39,24}$	$V_{24,4}$	$V_{4,16}$	$V_{16,5}$	$V_{5,39}$	$V_{39,16}$	$V_{16,23}$
μ	−30.18	−56.68	−8.61	8.43	128.60	46.52	−72.38	262.47	−30.16
σ	1.33	1.27	3.44	3.92	5.19	2.2	0.73	10.82	1.81

On the other hand, from the observation of the collected results, we clearly see that there are some transformations that would likely result in a notable slowdown in most programs like pass number 16 (Alias Analysis pass). However, others have the opposite effect, resulting in a speed up in most cases like pass number 24 (Lower Expect Intrinsic). More interestingly, the rest of the transformations resulted in speed up in some cases and slowdown in others like transformation number 34 (loop unswitch), confirming our prediction that the effect depends on the characteristics of the input program, hence comes the need for the profiling step.

Another idea worth investigating is the effect of the input data size and characteristics on the selection of the obfuscation sequence. We experimented with the 401.bzip2 benchmark (also from SPEC CPU 2006 suite), with different data sizes, the first is the small input obtained from the standard test data set provided by SPEC and the other bigger input is from the standard reference data set. For that purpose, we collected the profiling data as shown in Fig. 1-g and 1-h, and the results were similar in both scenarios. Therefore, the candidate passes would be also similar. However, when applying our iOJIT system, due to the randomness in the algorithm, the resulting obfuscation sequences were different as shown in the last 2 rows in Table 1. Since most of the passes are control-flow transformations, we suspect that either way, the selected sequence would be effective regardless of the input data size, but such claim will be further investigated in future work.

Figure 3 shows a consecutive comparison between 9 obfuscated code versions the *compress* function from the 401.bzip2 benchmark according to our software complexity metrics we mentioned in Sect. 2. These code versions were generated by applying a pair of transformations from the candidate passes obtained from Table 1 {6,7,4,12,25,32,22,28,17,11 } taken as pair by pair, i.e. using sequences of length of 2. That is version one is the product of transformation pair (6,7), version two is produced by (7,4) and so on. The version of (0,6) was neglected, since it just means applying transformation 6, because the number 0 stands for applying no transformation at all. Then, each version was compared with the previous one starting from the original code. First, we measured the code similarity using the MOSS system shown in blue bars. Secondly, we illustrated how the number of branch instructions changed across these code versions as shown in the green line. Thirdly, the red lines shows how the total number of instructions in these code version changed as well. The Figure shows that each code version is quite different from the previous one even for that short sequence. Finally, we compared these results with the ones obtained using the old OJIT, detailed in [2], where we did similar experiments also on the compress function of the 401.bzip2 benchmark and the different code versions were the product of a randomly selected set of transformation passes. We found out that the overall performance of the new system is better in terms of execution time variations (−15 to 30% vs. −66.22 to 132.4%). Even for the code similarity metric for the *compress* function, 55 to 99%, the results are acceptable for such a small length sequence as opposed to the variable length used in [2]. Still, we expect the results to be improved if we applied randomly length-

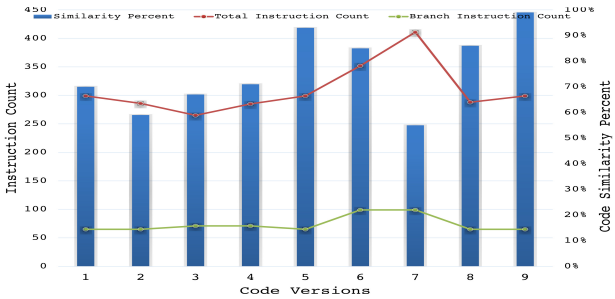


Fig. 3 Comparing different code versions of the compress function from the 401.bzip2 benchmark.

ened sequences of transformations.

In addition, in order to verify that the online performance of the system is stable when compared with the profiling phase, we ran the experiments on the 401.bzip2 benchmark over the course of 50 times with the big input size. Then we collected the $\Delta T\%$ statistics for the code version produced by the selected set of transformations. Our findings in Table 2 show that the mean values for $\Delta T\%$ noted as μ in Table 2, are comparable to the $\Delta T\%$ collected in the profiling phase in the last row of Table 1 for the corresponding code versions (i.e. neglecting the first column values in Table 1, which corresponds to the single transformation case, discussed above). In addition, the resulting standard deviation values, noted as σ in Table 2, are also within reasonable margins, meaning that our measurement error is almost negligible. Therefore, we can claim that the obfuscation sequence generated in the profiling phase is suitable across runs.

7. Conclusion and Future Work

In this paper we introduced an enhancement to an earlier system that uses obfuscation through JIT compilation to thwart side-channel attacks. In particular, we added a profiling step and an optimization algorithm to help choose the more efficient obfuscation transformations in accordance with the input program characteristics. We tested our system on a cloud platform running a variety of standard benchmarks from the SPEC CPU 2006 suite. Our newly introduced system achieved improvements, when compared with the old OJIT, in terms of execution time and code appearance changes yet retaining the system's dynamic randomness feature, rendering the code unintelligible for reverse engineering via statistical analysis or decompilation tools.

For future work, we plan to employ smarter selection algorithms, perhaps in the realm of machine learning. In fact, we hope be able to eliminate the need for the profiling phase. This would enable us to obfuscate the code on a finer-grain (function or module) level. We also plan to test the iOJIT in real attack scenarios. We also have plans to protect the compiler itself against tampering.

References

[1] D. Zissis and D. Lekkas, "Addressing cloud computing security issues," Future Generation Computer Systems, vol.28, no.3, pp.583–592, 2012.

[2] M. Hataba, A. El-Mahdy, and E. Rohou, OJIT: A novel obfuscation approach using standard just-in-time compiler transformations, 4th Int. Workshop on Dynamic Compilation Everywhere, 2015.

[3] M. Hataba, R. Elkhouly, and A. El-Mahdy, "Diversified Remote Code Execution Using Dynamic Obfuscation of Conditional Branches," IEEE 35th International Conference on Distributed Computing Systems Workshops, pp.120–127, 2015.

[4] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," 16th ACM conference on Computer and communications security, pp.199–212, 2009.

[5] S. Chari, C.S. Jutla, J.R. Rao, ad P. Rohatgi, "Towards Sound Approaches to Counteract Power-Analysis Attacks," 19th Annual International Cryptology Conference, vol.1666, pp.398–412, 1999.

[6] C. Collberg, C. Thomborson, and D. Low, A taxonomy of obfuscating transformations, Tech. Rep., no.148, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[7] The LLVM Compiler Infrastructure, http://www.llvm.org/, accessed Jan. 10, 2018.

[8] J.V. Cleemput, B. Coppens, and B. De Sutter, "Compiler mitigations for time attacks on modern x86 processors," ACM Transactions on Architecture and Code Optimization, vol.8, no.4, pp.1–20, 2012.

[9] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," 30th IEEE Symposium on Security and Privacy, pp.45–60, 2009.

[10] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity," Network and Distributed System Security Symposium, 2015.

[11] H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P.-C. Yew, "Control flow obfuscation with information flow tracking," 42nd IEEE/ACM International Symposium on Microarchitecture, pp.391–400, 2009.

[12] T. Toyofuku, T. Tabata, and K. Sakurai, "Program obfuscation scheme using random numbers to complicate control flow," International Conference on Embedded and Ubiquitous Computing, vol.3823, pp.916–925, 2005.

[13] D.E. Knuth, Seminumerical Algorithms, The Art of Computer Programming, vol.2, Addison-Wesley, 1969.

[14] MOSS: A System for Detecting Software Similarity, https://theory.stanford.edu/~aiken/moss/, accessed Jan. 10, 2018.

[15] Standard Performance Evaluation Corporation (SPEC) Benchmarks, http://www.spec.org/, accessed Jan. 10, 2018.

[16] R. Daudigny, H. Ledig, F. Muller, and F. Valette, "SCARE of the DES," International Conference on Applied Cryptography and Network Security, vol.3531, pp.393–406, 2005.

Appendix A: iOJIT Transformations

The iOJIT system uses a set of transformation as shown in the following table. We label each transformation with a number, that was referenced earlier in the paper. For more information on their operations, please refer to [7].

Table A·1 List of used transformations.

#Trans.	Pass Name	#Trans	Pass Name
1	Basic Alias Analysis	2	CFG Simplification
3	Reassociation	4	GVN pass
5	DCE	6	Constant Propagation
7	Instruction Combining	8	Aggressive Dead Code Elim.
9	Mem. Copy Optimization	10	Promote Mem. To Reg.
11	Dead Instruction Elimin.	12	Indirect Variable Simplify
13	LICM	14	Sinking
15	Instruction Namer	16	Alias Analysis Evaluator
17	De-linearization	18	Partially Inline Lib. Calls
19	Tail Call Elimination	20	Unreachable Block Elimin.
21	Break Critical Edges	22	Early CSE
23	GVN with (false) input	24	Lower Expect Intrinsic
25	Lower Invoke	26	Instruction Simplifier
27	Jump Threading	28	Stack Protector
29	Scalar Replicate Aggregates	30	Flatten CFG
31	Verfier	32	Demote Reg. To Mem.
33	Lower Switch	34	Loop Unswitch
35	Loop Rotate	36	Loop Deletion
37	Loop Unroll	38	Dead Store Elimination
39	SCCP	40	GC Lowering
41	Dwarf EH		