2015 IEEE 35th International Conference on Distributed Computing Systems Workshops

Diversified Remote Code Execution Using Dynamic Obfuscation of Conditional Branches

Muhammad Hataba*, Reem Elkhouly* and Ahmed El-Mahdy*
* Parallel Computing Lab, Computer Science and Engineering Department
Egypt-Japan University of Science and Technology (E-JUST), Alexandria, Egypt
Emails: {mohamed.hataba,reem.elkhouly,ahmed.elmahdy}@ejust.edu.eg

***Abstract*—Information leakage via timing side-channel attacks is one of the main threats that target code executing on remote platforms such as the cloud computing environment. These attacks can be further leveraged to reverse-engineer or even tamper with the running code. In this paper, we propose a security obfuscation technique, which helps making the generated code more resistant to these attacks, by means of increasing logical complexity to hinder the formulation of a solid hypothesis about code behavior. More importantly, this software solution is portable, generic and does not require special setup or hardware or software modifications. In particular, we consider mangling the control-flow inside a program via converting a random set of conditional branches into linear code, using if-conversion transformation. Moreover, our method exploits the dynamic compilation technology to continually and randomly alter the branches. All of this mangling should diversify code execution, hence it becomes difficult for an attacker to infer timing correlations through statistical analysis. We extend the LLVM JIT compiler to provide for an initial investigation of this approach. This makes our system applicable to a wide variety of programming languages and hardware platforms. We have studied the system using a simple test program and selected benchmarks from the standard SPEC CPU 2006 suite with different input loads and experimental setups. Initial results show significant changes in program's control-flow and hence data dependences, resulting in noticeable different execution times even for the same input data, thereby complicating such attacks. More notably, the performance penalty is within reasonable margins.**

Keywords*—*Obfuscation, Side-Channels, JIT Compilation, If-Conversion.

I. Introduction

Computation as a service utility is currently a popular trend. With the evolution in web services, computational hardware and communication, remote execution platforms such as cloud computing are becoming more ubiquitous. However, nowadays security breaches are becoming more advanced and structured and their prime target is major computing service providers. Being one of the most popular technology services, cloud computing platforms are the honeypot of many wide scale security attacks such as intrusion, hijacking authentication, Denial of Service (DoS), data theft and/or sabotage, racketeering and our subject of concern, side channel attacks.

A major advantage of the cloud delivery model is the rapid on-demand provisioning of computing resources, that exploits economy of scale, allowing for minimal management costs [27]. To realize such abstractions, cloud computing inherently depends on the virtualization technology to hide the complexity of managing the hardware to the provider side [6]. However, this virtualized environment is susceptible to security vulnerabilities, since the physical computing resources are shared amongst different virtual machines, most likely for different users, consequently existing security measures are not enough [21].

In particular, existing cryptographic techniques that rely on digital signatures, certificates or trusted platforms [24], are not best suited for these platforms as the decryption process happens remotely, making it potentially possible for attackers to observe the actual running process, exposing encryption keys [41]. Homomorphic encryption is a relatively new approach for encrypted execution; however, it is not currently practical due to its special setup requirements and tremendous application cost [36]. Therefore, new practical security defenses are sought to protect the execution of programs from potential adversaries sharing the same physical hardware.

An earlier approach [22] has suggested the usage of Just-In-Time compilation (JIT) to conceal program execution via some obfuscation transformations. Their rationale is that compilers by definition maintain the semantics of programs, while applying various optimizations to the code. Moreover, the dynamic nature of JIT can provide for continually obfuscating the code, making it difficult for adversaries to monitor or otherwise exploit the execution trace.

In this paper, we focus on disrupting the control-flow of a program by dynamically changing the conditional branch instructions, while retaining the same program functionality. In particular the paper has the following contributions:

- We present a portable software-based security solution to protect code running on remote platforms against side-channel attacks.
- We extend the LLVM compilation infrastructure to allow for random yet dynamic application of if-conversion transformations on conditional branches.
- We illustrate the effect of our proposed obfuscation scheme on the diversification of the control-flow graph and consequently on the total execution times for a set of standard benchmarks borrowed from the SPEC CPU 2006 suite.

The remainder of this paper is organized as follows: Section II describes the threat model for remote execution environments. Section III summarizes control-flow obfuscation techniques with a particular focus on conditional branches conversion. Section IV proposes our system implementation. Section V presents our initial experimental results. Section VI surveys some related work. Section VII shows directions for future work. Finally, Section VIII concludes the paper.

II. Threat Model

Side-channel attacks are of the most serious security breaches that come with cloud computing platforms. An intruder would implant himself virtually inside a cloud provider's physical resources. Hence, he could harness the surrounding resources for information leakage. The attacker can also be a malicious insider or an untrusted system operator trying to launch these attacks. These side-channels can manifest in various forms depending on the attack surface. There are access-driven attacks that analyze the access patterns of memory, processor registers, storage or other physical resources and try to find the correlation there [30]. More recently there are acoustic attacks, which exploit the sounds from computer hardware such as the processor to recognize its operations [20]. However, these attacks are more difficult to launch since attackers needs to be in physical proximity of the victim hardware, a situation unlikely to happen in a cloud setup unless it's a malicious insider who is doing the attacks. Further attacks that we consider are timing attacks [10], where an attacker monitors the execution time of a running program and then tries to infer knowledge about the behavior of the program. Our measurements show how our system can resist these attacks by continuously changing the execution time for multiple runs of a program even for the same input. In addition to that, we also consider the more advanced trace-driven attacks, which analyze execution-trace of the program in order to reverse engineer it [1]. Section V demonstrates how we diversify the control-flow graph of a program as a defense against such threats.

There is a great deal of research work in the literature as to how to reconstruct the original structure of a program from the collection of its trace. Reverse engineering could have the ramifications of unauthorized tampering or infringement of intellectual property. The binary form (machine code) could be disassembled into assembly code and afterwards possibly decompiled to get the higher-level representation [14]. The decompilation of a program needs to meticulously study the program statically in order to understand it. Analysis could be done by pattern matching, slicing the program variables, partial evaluation and automated theorem proving of some control statements [16].

In [32], the authors presented an analytical study of such threats. They launched large scale cartographic attacks to model placement information of virtual machines (VMs) in cloud computing services platforms. Their goal was to effectively choose where to implant their malicious virtual machine in co-residence with a certain victim in order to launch cross-VM side channel attacks. To show how serious their work, they experimented on Amazon's EC2 cloud and achieved a success ratio of 40%.

III. Control-Flow Obfuscation

Our proposed method to protect programs is incorporating diversification and randomization to increase logical complexity. That is obscuring system internals from adversaries as opposed to the orthodox open security that rely on key-based cryptography to protect computing systems [34]. Obfuscation is widely adopted in virus and malware designs to hide code signatures from scanners especially when implanted in remote platforms [44]. Obfuscation has been investigated earlier [7] and some researchers even redeemed it impossible [8], but nowadays it has gained more popularity [19]. In summary, code obfuscation techniques can be classified according to the transformation subject as follows. First, we have the high-level transformation of code layout, which can be changed during pre-processing phase. Furthermore, there are low-level physical transformations, which change machine code instruction and/or register and memory addressing. On the other hand, data transformations change the program variables and data structures into unusual representations. Finally, control-flow transformations were introduced to change the apparent control dependences in a program via various manipulation techniques such as loop unrolling/switching, function inlining/outlining, opaque predicates, dummy tasks and branch conversions. Here we focus on the latter control-flow obfuscations that could be promoted to architecture specific transformations. Conditional branches are the main reason for control dependences in a program. Normally, these branches impose a major cost in a program's runtime. Consequently, conditional branches present hindrances to parallelism or pipelining efforts [3]. Generally speaking, up until now many researchers often tried various techniques to convert or resolve branches for the sake of performance improvements [37]. Hence, speculation was introduced to minimize the effect of such control dependence while still keeping the correct data flow. Speculative execution uses branch prediction to guess the address of next instruction to be executed apriori then fetch, issue and start executing that instruction as if our branch predictions were always correct [23]. If a misprediction is discovered at runtime, the whole pipeline is flushed, the correct instruction is fetched and any performed operation is wasted. On the other hand, the authors in [26] tried to alleviate such control dependences by the use of predicates, which are a type of guarded instructions. The idea behind it is to decide whether to execute an instruction or not according to the evaluation result of some guard expression.

Additionally, there are many techniques introduced for the optimization of conditional branch. First, we have "Branch Relocation Optimization"; which makes the branches and their targets in the same loop nesting level. This can be done by converting exit branches – the ones that has a target outside loop nesting level; into forward – target after branch – or backward branches – the target before the branch – with a target in the same nesting level [4]. Second, there is "Branch Removal Optimization"; which removes forward branch by adding guard expressions [4]. Finally, there is "Conditional Moves (CMOVs)"; which is a hardware enabled copy of a value from a location to another only when a condition is met where its result is stored in a register. That is converting the control dependence into data dependence, which changes the control-flow of a program by eliminating the branching. These types of instructions are hardware specific and were introduced since Intel's Pentium Pro in 1995 [11].

Here, we revisit the problem of deciding which branches to convert (or otherwise replace their subsequent control dependence with data dependence) from the security prospective. That is, We utilized the branch optimization techniques for the sake of security through obscurity by varying their time cost

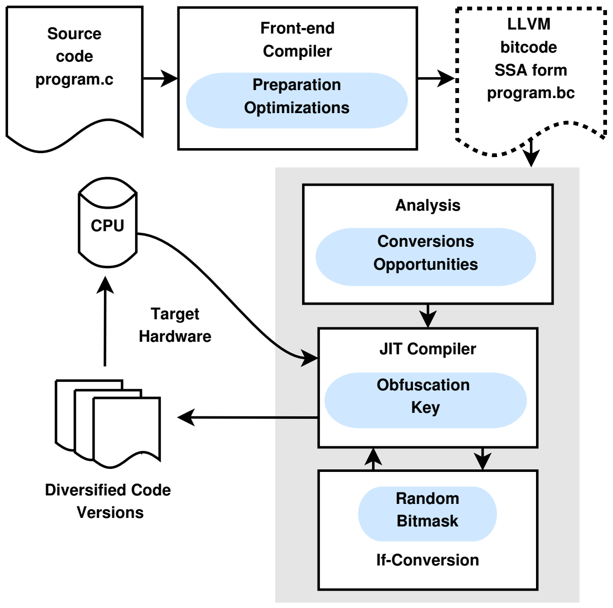


Fig. 1: System architecture diagram.

to thwart probabilistic analysis of code execution. Moreover, we randomly and dynamically convert conditional branches to keep our program obfuscated at runtime to provide for code diversity, hopefully, at a low cost of its overall performance.

IV. System Implementation

Our implementation is based on the LLVM infrastructure [39], which is a compilation framework that supports a wide range of high-level programming languages and an even larger set of processor architectures. LLVM provides different analysis and optimization plugins (called passes) that can be called at compile time, link time, runtime or even between runs. To simplify such facilities; LLVM relies on Static Single Assignment (SSA) code representation, where a variable is assigned only once.

LLVM also offers dynamic compilation via the Just-In-Time (JIT) compiler, where program units (such as functions or basic blocks), are compiled on-demand, at the execution time, yielding a generated machine code that is target dependent. Hence, to better serve our obfuscation goals, we choose to apply branch conversion dynamically during runtime. Having the program compiled by the LLVM JIT compiler, we instructed the compiler to instantaneously generate random sequences of bit-masks to control the conversion of conditional statements, thereby, changing the original control-flow of the program and hence its execution time

In our work, we override the normal behavior of LLVM transformation pass, which is responsible for the if-conversion optimization, to implement our randomized control-flow obfuscation system prototype. When applied to a program, it discovers the if-conversion opportunities within each function. Then, a *bit-mask* is randomly generated as a sort of an obfuscation key that manages converting a different set of branches within each compilation phase. This bit-mask is the key to our continuous diversification process and is produced dynamically at runtime. For each function in the program, the bit-mask is structured as a string of bits equal to the no. of branches within the target function, where each of them tells the compiler whether or not to convert the corresponding branch. Consequently, this code diversification process generates different versions of program pieces having dissimilar control-flow graphs.

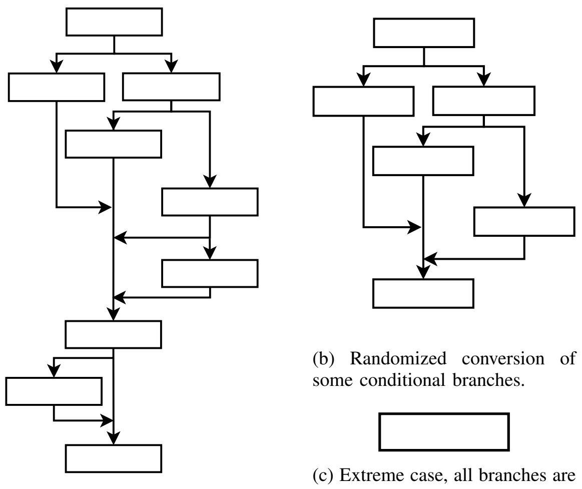
We do so via dynamic compilation via the Just-In-Time (JIT) compiler, where program units (such as functions or basic blocks), are compiled on-demand, at the execution time, yielding a generated machine code that is target dependent. Hence, to better serve our obfuscation goals, we choose to apply branch conversion dynamically during runtime. Having the program compiled by the JIT compiler, we instructed the compiler to instantaneously generate random sequences of bit-masks to control the conversion of conditional statements, thereby, changing the control-flow of the program and hence its execution time.

Figure 1 depicts the operational processes of the prototype of our system. In this implementation, we build a module inside the LLVM compiler infrastructure. At the beginning, our system inspects the input program, so as to determine the opportunities for if-conversion within each function. In order to maximize these opportunities, we apply some preparation optimization passes to the input program after it has been converted to LLVM intermediate representation (often called *Bitcode*). These operations include a sequence of analysis and transformation passes to simplify loops and induction variables. Then, we may apply loop unrolling more efficiently. Finally, we transform all memory references to register form.

Now that the code is geared up in SSA form; we apply an analysis phase to calculate the conversion opportunities (n) in every function (which is the default compilation unit). For each of which we could generate up to 2^n different bit-masks. Every time a piece of the program (a compilation unit) is being compiled using LLVM's JIT compiler, a new bit-mask is randomly generated and applied during runtime; effectively producing a diversified code version with the same functionality. This trampoline action between the compiler and program execution can be modified to happen in accordance to our obfuscation needs. For example, if we want to compile a recursive function –which could be a hotspot in some program– differently every time producing a different versions from it (this is also code morphing [2]) with exactly the same functionality. Hence, control dependence and execution trace become very hard to exploit or otherwise comprehend by an attacker due to the increased logical complexity. Even monitoring execution timing patterns –suspiciously via launching side-channel attacks– would be quite unyielding because of the difficulty of coming with probabilistic correlations. The hypothesis of such an attack scenario will be investigated in future work.

V. Experiments and Evaluation

In this section, we put our system into test in real execution scenarios with varying workloads to assert the validity of our hypothesis. The permutations of control-flow graph (CFG) that



(a) Original control-flow graph.

(b) Randomized conversion of some conditional branches.

(c) Extreme case, all branches are linearized [single basic block].

Fig. 2: Three control-flow graphs for the same program.

Algorithm 1 Pseudo code of a simple test program.

```
$a \leftarrow val1$, $b \leftarrow val2$, $c \leftarrow val3$
**if** $param > Th1$ **then** $a \leftarrow newVal1$, $b \leftarrow newVal2$
**else**
    **if** $param > Th2$ **then** $a \leftarrow newVal3$
    **else**
        **if** $param > Th3$ **then** $a \leftarrow newVal4$
        **end if**
        $b \leftarrow newVal5$
    **end if**
    $c \leftarrow a$
**end if**
**if** $param < Th2$ **then** $c \leftarrow newVal6$
**else**$c \leftarrow newVal7$
**end if**
**return** $a + b + c$
```

were introduced using our obfuscation technique is explained on a simple program. Then, the diversification of the programs' behavior at runtime, due to dynamic randomization of branch conversion, is discussed through experimentation below.

A. Control-Flow Obfuscation Study of a Simple Program

Our randomized control-flow obfuscation technique, when applied to a program, generates diversified versions of the program each with different control-flow graph. That is due to the changes it applies to the branches within the code. Converting a different set of branches along with every compilation step causes this effect. That helps to hide the code behaviour from an adversary on the remote execution environment. As an example, we considered a simple program with the pseudo code in Algorithm 1. As shown, the program contains multiple if-statements with various branching factors and nesting levels. Figure 2 shows the obvious change in the control-flow graph of original and diversified versions of the same program. The original program before obfuscation is represented by the control-flow graph shown in Figure 2a. When the randomized control-flow obfuscation technique is applied, we get a diversified control-flow graph such as the one shown in Figure 2b. There, it is noticed how some branches are replaced by serial code which change the control-flow graph. Eventually, an extreme control-flow obfuscation resulted in a control-flow graph contains a single basic block as shown in Figure 2c. Through these figures, we can estimate the potential for control-flow graph diversification to be an effective obfuscation technique if applied dynamically and randomly enough.

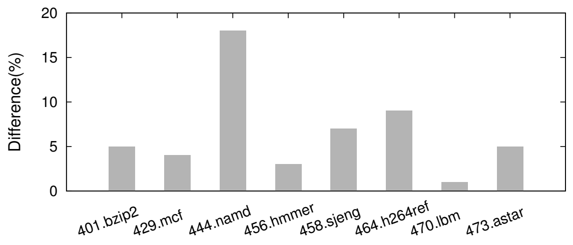


Fig. 3: The percentage of code difference between the original program and AC version for each benchmark.

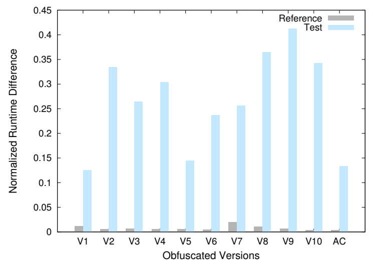
B. Diversity of Runtime due to Control-Flow Obfuscation

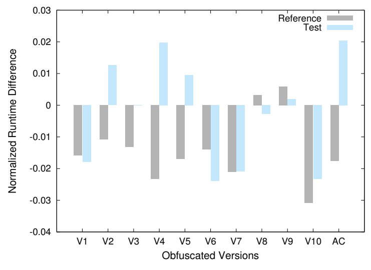
For the compiler infrastructure we modified LLVM compilation framework version no. 3.6 with our obfuscation extension. As for the hardware, we ran these programs on a machine equipped with a 2 GHz *Intel Core i7* processor running with 4GB of RAM; noting that current LLVM compilation framework support a large set of architecture ranging from the *ARM* processors, which are widely popular in smartphones to other CPUs such as *SPARC, PowerPC, AArch64 ...etc.*. It is worth noting that this set up is just a testbed to show the existence of promising results as per some obfuscation metrics [21], [5]. We tested different versions of the same running program to verify that there are randomly unpredictable control-flow disruptions happening dynamically during runtime, which are manifested in the execution trace of the program and consequentially in its execution time resulting in notable differences. Hence, we suspect that an adversary eavesdropping on a running program on a remote system would not be able to inspect or reverse engineer the original source code from the collection of such uncorrelated timing observation. We selected an arbitrary set of programs from the SPEC CPU 2006 v.1 [38] benchmark suite and subjected them to our proposed technique that is dynamic obfuscation of conditional branches during the JIT compilation phase. Namely, we operated on the following benchmarks: *401.bzip2, 429.mcf, 444.namd, 456.hmmer, 458.sjeng, 464.h264ref, 470.lbm* and *473.astar* just as a sample show case. Nevertheless, our system is general enough to accommodate any program that contains conditional branch instructions.

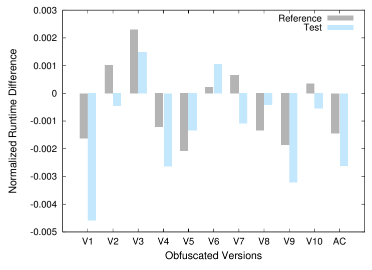
As we mentioned before, the morphing pool of various program versions is very huge in the order of (2^n) possible variations, where (n) is the number of conditional branches that can be *if-converted*. Therefore, in our experiment we only ran each test program for 12 times starting from a base line original version with non branches converted, reaching up to an extreme version with all branches converted (abbreviated **AC**), and ten versions with random bitmasks in between these two, that is just to show case our system in work and

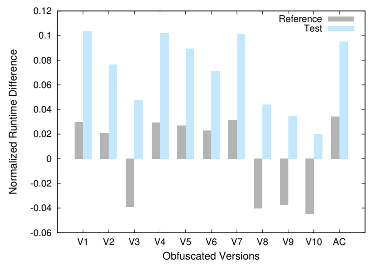
Benchmark	401.bzip2	429.mcf	444.namd	456.hmmer	458.sjeng	464.h264ref	470.lbm	473.astar
Original code size	20674	2968	79345	78583	36576	158703	2648	12969

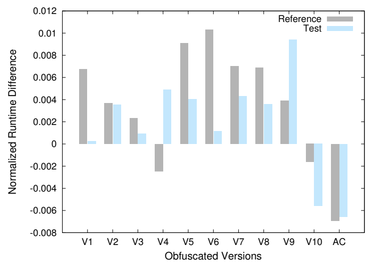
TABLE I: No. of code lines in each program.

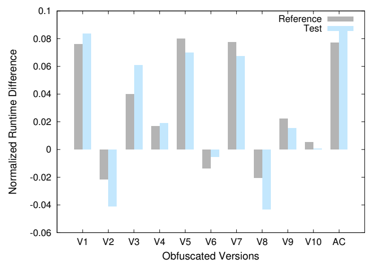

(a) 401.bzip2

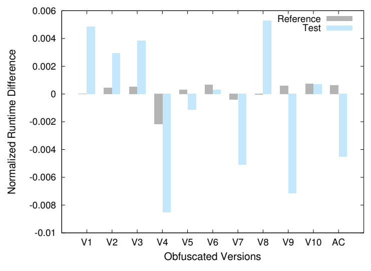

(b) 429.mcf


(c) 444.namd


(d) 456.hmmer


(e) 458.sjeng


(f) 464.h264ref


(g) 470.lbm

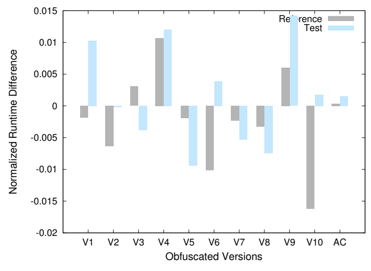

(h) 473.astar

Fig. 4: Timming difference of obfuscated versions normalized to original program's runtime.

to highlight the boundaries. To understand the effect of if-conversion obfuscation, we analyze the results both statically and dynamically as follows.

*1) **Static Analysis**:* The generated machine code of diversified if-converted versions was thoroughly inspected and compared to the original program. Among the versions of the same program, instruction count (IC) has changed slightly, nevertheless, when examining the code appearance, we noticed great changes between these code versions. Here, we investigated the difference between the original and AC versions of the same program using a standard system for detecting software plagiarism called (MOSS: Measure Of Software Similarity) [33], [29]. We found out that the the code difference is proportional to the number of conditional branches and inversely proportional to the code size which can be obviously seen when comprehending Figure 3, Table I and II together. For example, the largest difference happens in the 444.namd, which contains relatively large number of conditional branches along with small code. On the contrary, the 429.mcf has the least difference with few branches and large code. The cause of such dissimilarities is the changes in the control-flow dependences and hence the data-flow dependences. Consequentially, trace driven side-channel attacks will be cumbersome and reverse engineering of the code is very difficult due the severe logical complexity of the code. We have to acknowledge the fact that for some programs, such as 470.lbm, there is a smaller margin for diversification (small percent of convertible branches) as opposed to 444.namd for example. This is reflected in their corresponding code difference percentages shown in Figure 3.

Benchmark	**Branch**	**V1**	**V2**	**V3**	**V4**	**V5**	**V6**	**V7**	**V8**	**V9**	**V10**	**AC**
401.bzip2	Triangles	9/35	13/35	15/35	12/35	8/35	8/35	18/35	12/35	10/35	13/35	35/35
	Diamonds	5/5	2/5	1/5	2/5	3/5	0/5	2/5	2/6	2/5	2/5	6/6
429.mcf	Triangles	10/12	6/12	3/12	4/12	6/12	5/12	4/12	3/12	5/12	5/12	12/12
	Diamonds	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
444.namd	Triangles	32/123	35/123	36/123	47/123	34/123	41/123	25/123	39/123	24/123	48/123	123/123
	Diamonds	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
456.hmmer	Triangles	47/137	38/136	45/137	54/138	53/140	33/137	37/136	48/138	46/139	41/136	147/149
	Diamonds	3/13	3/11	6/12	3/11	2/11	5/12	9/13	5/13	5/12	3/9	15/15
458.sjeng	Triangles	42/153	40/155	47/155	28/152	27/155	30/153	37/152	34/152	18/152	53/154	167/167
	Diamonds	0/2	1/3	2/4	1/4	1/3	2/4	1/3	2/3	2/4	0/3	5/5
464.h264ref	Triangles	108/441	97/440	111/432	95/434	95/434	119/439	108/439	86/429	100/438	91/437	515/515
	Diamonds	2/121	6/121	9/120	4/120	5/120	10/120	14/119	12/121	10/120	9/121	136/136
470.lbm	Triangles	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
	Diamonds	1/1	0/1	1/1	1/1	1/1	1/1	1/1	0/1	1/1	0/1	1/1
473.astar	Triangles	15/52	10/52	16/52	11/52	10/52	17/52	15/52	13/52	23/52	10/52	55/55
	Diamonds	0/1	0/1	0/1	0/1	0/2	0/1	0/2	1/2	1/2	0/1	2/2

TABLE II: No. of converted/total of triangle or diamond conditional branches in obfuscated versions.

Benchmark		**Test Input Statistics**				**Reference Input Statistics**			
		JIT Compile Time		**Execution Time**		**JIT Compile Time**		**Execution Time**	
		Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.
401.bzip	Obfuscated	0.95667	0.00918	0.02675	0.00360	0.96081	0.00919	4.97719	0.02269
	Original	0.95051	0.00661	0.03589	0.00695	0.94958	0.00487	5.01432	0.02945
429.mcf	Obfuscated	0.14072	0.00258	3.01970	0.05203	0.13780	0.00157	455.31862	4.83647
	Original	0.13980	0.00091	3.00770	0.07157	0.13687	0.00114	449.45793	3.74004
444.namd	Obfuscated	3.97365	0.01837	35.44418	0.06173	3.98203	0.02310	456.70113	0.62827
	Original	3.96413	0.02703	35.39887	0.06445	3.96663	0.02945	456.46797	0.15391
456.hmmer	Obfuscated	3.65708	0.03987	4.02425	0.13353	3.64601	0.02022	526.48374	18.64259
	Original	3.35310	0.01475	4.31830	0.14196	3.43831	0.17698	526.19939	18.01393
458.sjeng	Obfuscated	1.65161	0.01182	4.37114	0.01939	1.64233	0.01040	681.19450	3.48622
	Original	1.62667	0.01103	4.37843	0.01605	1.62519	0.02030	683.52731	1.79690
464.h264ref	Obfuscated	8.25360	0.04188	15.28707	0.04188	8.27397	0.04110	89.67445	3.71027
	Original	7.24541	0.02994	15.70119	0.04069	7.26593	0.03698	92.30027	0.32528
470.lbm	Obfuscated	0.12358	0.00418	2.38858	0.01116	0.11813	0.00206	306.22104	0.24445
	Original	0.12527	0.00497	2.38683	0.01411	0.12108	0.00622	306.25372	0.25502
473.astar	Obfuscated	0.53138	0.00677	10.99879	0.08691	0.52160	0.00364	219.09940	1.64150
	Original	0.50549	0.00495	11.02331	0.09583	0.50188	0.00586	218.82642	1.40680

TABLE III: Timing statistics of the obfuscated and original versions of the selected benchmarks with different input loads.

2) ***Dynamic Analysis:*** The most important feature of our work is the random changes in the code runtimes during the actual execution of the programs. For each program, we studied 10 diversified versions generated by our system while emitting different random bit-masks to control the if-conversion process, plus the AC version. The results were normalized and compared with the observations collected from the corresponding program's original version with no obfuscation, all having the same input data. Figure 4 shows the variations in the runtime of if-converted versions of each benchmark as opposed to the original program's runtime. The measurements were collected under two standard workload –borrowed from the SPEC suite– which differ in size; the test input data (smaller one) and the reference input data (considerably larger, which eventually translate into more runtime). In this figure, we can see a range of variation approaches 40%, $\pm$3%, $\pm$0.4%, $\pm$11%, $\pm$1.1%, $\pm$9%, $\pm$0.8% and $\pm$1.5% in 401.bzip2, 429.mcf, 444.namd, 456.hmmer, 458.sjeng, 464.h264ref, 470.lbm and 473.astar respectively. These result are aided by the observation from (Table II), which analyzes how much control-flow obfuscation was introduced in terms of conditional branches conversion –either triangle branches (simple if statement) or diamond branches (having an else part) – in each code version. It's fairly noted the programs that responded well to our obfuscation mechanism are the ones that actually have more candidate branches, i.e. more opportunities for obfuscation. Also, there is no simple or generalized correlation between the number of branches converted and the actual improvement in runtime; hence such optimization problem is the subject of wide research topics [4], even predicting the opposite, which is the worst case execution time (WCET) of programs, exactly in Real-time is very complicated [43]. Another remark from the Figure 4 is that changing input size would have unpredictable repercussion on the observed runtime even for the same obfuscated code version. Some benchmarks may outperform the original version in terms of execution time, others may be slower. This is quite apparent in programs like 429.mcf, 456.hmmer, 458.sjeng and 473.astar. Other programs like 401.bzip2 and 464.h264ref are quite proportional to the input size. This is logical because theses program performs file and video compression respectively. This proves that it's hard for an attacker to analyze code behaviour even under some fixed circumstantial settings.

It is important to notice from the statistics collected in Table III that the compilation overhead due to obfuscation is minimal and the overall JIT compilation time is basal compared to the total execution time. In addition to that, although the time variance of multiple runs of the original code is negligible; for the obfuscated versions, the execution time deviations under different workloads is quite promising, especially of course for those programs that have more branches and consequently more diversification opportunities, such as the 464.h264ref benchmark. Moreover, these statistics can be further analyzed thoroughly to give an approximate estimation about the timing bounds of the program execution.

Nevertheless, the effectiveness of our obfuscation scheme can be utilized to conveniently target some specific parts of an entire program, which may be designated as security critical spots. These parts can be augmented with opaque predicate to make them more suited for diversification. Moreover, we can change the default behavior of the JIT compiler to make it flush the cache and recomiple code again with new branch labels and target addresses. This can be done by deleting compilation stubs and symbol, thus adding more confusion on snooping attackers and preventing cache information leakage via side-channels. As the for the rest of the program where no branch conversion can be introduced, we suggest to incorporate other obfuscation techniques such as [22].

VI. Related Work

Although most of the ideas we utilized in our system have been investigated before in some way or another (e.g. if-conversions, or code obfuscation during runtime); the novelty is in tying the right components together to better serve our security needs without making the system overly complex. That been said, there are various techniques to mitigate side-channel attacks in the literature [15], [17], [28] and more practically the work of Burket et al [12]. But most of these techniques study the problem of key protection, that is eliminate the assumption that a program's behavior would depend on input data size. For example, in [17], aiming to protect the key, the authors eliminated all conditional branches that depend on the key, thus the control-flow is the same regardless the input(i.e. key independent). This technique is sometimes called isochronous coding. That is different form our method, which tries to protect a program from behaviour analysis or reverse engineering during runtime by disrupting the program's control-flow to avert attackers from recognizing which operation is being performed at a certain point of time. Yet, our method could serve the same goal, since there is no apparent correlation between the input size and the randomized control-flow we produce, therefore, a hypothesis can hardly be made about the key size. This claim will be further investigated in future work.

It is worth mentioning that code morphing was suggested in previous work such as [2], [9] to counter differential power analysis (DPA) attacks, or to thwart cache side-channels as in [18]. Nevertheless, our proposed technique depends on simple generic transformations rather than inserting random delays and bogus code, which sometimes may be difficult. Also, changing the conditional branches in our code would combine the protection against cache attacks; if the memory layout were to be changed at runtime randomly by the recompilation scheme that we discussed in the previous section. Additionally, our method provides a dynamic protection against DPA, by disrupting the time dimension, which an attacker relies on to build his analysis formulation by monitoring the operational flow of the program from measurements of power levels.

While there exist several techniques for control-flow obfuscation [16], [40], [13], [35] and more precisely branches obfuscation [42], [25], [31] generally tailored to impede static disassembling, our method has the advantage of being dynamically and randomly changing utilizing the dynamic nature of LLVM's JIT compiler. Moreover, the use of such open source framework offers the applicability of our solution to a wider range of high level languages and architectures. All of that makes our system more suitable for application on remote and often hostile computing environments such as the cloud computing, especially in the lack of precise information about the target platform or operational circumstances to protect against.

VII. Future Work

For future work, we plan to optimize the obfuscation process for better results in terms of increased diversification and logical complexity to further confuse attackers. One possibility is to select specific branches for conversion that would generate larger variation in the execution time. Another idea is perhaps the integration with other security techniques and obfuscation mechanism to cover a wider range of attack vectors. Adding controllability for the user to tradeoff security for performance would be advantageous, since this is the key idea of such remote computing service utilities. Implementation of side-channel attack scenarios is currently studied, combined with a thorough analysis of our system's strengths and weaknesses. Another important task is securing the JIT compiler itself (possibly by self obfuscation) and also investigating the delivery model of the obfuscated code to the remote platform along with protecting secret data as well.

VIII. Conclusion

The new paradigm shift of remote execution platforms introduced new security issues and trust doubts. Side-channel attacks present a serious threat for these platforms. This paper presents an initial investigation for the use of a proposed control-flow obfuscation method, which is generic and easily applicable. We dynamically disrupt the control-flow of a program by converting its conditional branches randomly. We implemented our technique on the LLVM JIT compilation infrastructure, thus our system is source language and platform independent. Moreover, we experimented on selected benchmarks from the standard SPEC CPU 2006 benchmarks suite. The preliminary result show noticeable changes in execution times due to code changes in data and control-flow dependences, potentially rendering the code unintelligible for reverse engineering via statistical decompilation analysis, hence side-channel attacks would be more difficult. Nevertheless, the overhead of applying our technique is minimal.

IX. Acknowledgement

The authors wish to acknowledge the fruitful discussions, advice and contributions from Prof. Kazunori Ueda, Prof. Keiji Kimura, and Prof. Hironori Kasahara, Waseda University.

References

[1] O. Acıiçmez and Ç. K. Koç. Trace-driven cache attacks on aes (short paper). In *Information and Communications Security*. Springer, 2006.

[2] G. Agosta, A. Barenghi, and G. Pelosi. A code morphing methodology to automate power analysis countermeasures. In *Proceedings of the 49th Annual Design Automation Conference*, pages 77–82. ACM, 2012.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, 1983.

[4] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*, volume 289. Morgan Kaufmann San Francisco, 2002.

[5] B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, and B. Preneel. Program obfuscation: a quantitative approach. In *Proceedings of the 2007 ACM workshop on Quality of protection*, 2007.

[6] R. J. Aumann and S. Hart. *Handbook of game theory with economic applications*, volume 2. Elsevier, 1994.

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

[8] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im) possibility of obfuscating programs. In *Advances in Cryptology-CRYPTO 2001*. Springer, 2001.

[9] A. G. Bayrak, F. Regazzoni, P. Brisk, F.-X. Standaert, and P. Ienne. A first step towards automatic application of power analysis countermeasures. In *Proceedings of the 48th Design Automation Conference*, pages 230–235. ACM, 2011.

[10] J. Bonneau and I. Mironov. Cache-collision timing attacks against aes. In *Cryptographic Hardware and Embedded Systems-CHES 2006*. Springer, 2006.

[11] R. Bryant and O. David Richard. *Computer systems: a programmer's perspective*. Prentice Hall, 2003.

[12] J. Burket and S. Gottlieb. If-conversion to combat control flow-based timing attacks. 2014.

[13] H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P.-c. Yew. Control flow obfuscation with information flow tracking. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[14] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software: Practice and Experience*, 1995.

[15] 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 (TACO)*, 2012.

[16] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[17] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Security and Privacy, 2009 30th IEEE Symposium on*, 2009.

[18] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *Proceedings of the Network And Distributed System Security Symposium, NDSS*, 2015.

[19] C. Edwards. Researchers probe security through obscurity. *Communications of the ACM*, 2014.

[20] D. Genkin, A. Shamir, and E. Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. Technical report, Cryptology ePrint Archive, Report 2013/857, 2013.

[21] M. Hataba and A. El-Mahdy. Cloud protection by obfuscation: Techniques and metrics. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2012 Seventh International Conference on*. IEEE, 2012.

[22] M. Hataba, A. El-Mahdy, and E. Rohou. OJIT: A novel obfuscation approach using standard just-in-time compiler transformations. In *The Proceedings of the 4th Dynamic Compilation Everywhere workshop held in conjucntion with 10th HiPEAC Conference*. ACM, 2015.

[23] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[24] D. C. Latham. Department of defense trusted computer system evaluation criteria. *Department of Defense*, 1986.

[25] 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*, 2003.

[26] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. Hwu. A comparison of full and partial predicated execution support for ilp processors. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, 1995.

[27] P. Mell and T. Grance. The nist definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2011.

[28] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology-ICISC 2005*. 2006.

[29] MOSS: A System for Detecting Software Plagiarism. https://theory.stanford.edu/ aiken/moss/. Accessed Apr. 10, 2015.

[30] M. Neve and J.-P. Seifert. Advances on access-driven cache attacks on aes. In *Selected Areas in Cryptography*. Springer, 2007.

[31] I. V. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *USENIX Security*, 2007.

[32] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[33] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[34] C. E. Shannon. Communication theory of secrecy systems*. *Bell system technical journal*, 28(4):656–715, 1949.

[35] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[36] N. P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography–PKC 2010*. Springer, 2010.

[37] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*. IEEE Computer Society Press, 1981.

[38] Standard Performance Evaluation Corporation. SPEC Benchmarks, http://www.spec.org/. Accessed Apr. 10, 2015.

[39] The LLVM Compiler Infrastructure. http://www.llvm.org/. Accessed Apr. 10, 2015.

[40] T. Toyofuku, T. Tabata, and K. Sakurai. Program obfuscation scheme using random numbers to complicate control flow. In *Embedded and Ubiquitous Computing–EUC 2005 Workshops*, 2005.

[41] M. Van Dijk and A. Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. *HotSec*, 2010.

[42] Z. Wang, C. Jia, M. Liu, and X. Yu. Branch obfuscation using code mobility and signal. In *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*, 2012.

[43] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al. The worst-case execution-time problemoverview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[44] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *BWCCA*, 2010.