IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 8, 2022, accepted March 7, 2022, date of publication March 14, 2022, date of current version April 1, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3159249

Enhanced Obfuscation for Software Protection in Autonomous Vehicular Cloud Computing Platforms

MUHAMMAD HATABA[1,2], (Member, IEEE), AHMED SHERIF[1], (Senior Member, IEEE), AND REEM ELKHOULY[3]

[1]School of Computing Sciences and Computer Engineering, The University of Southern Mississippi, Hattiesburg, MS 39401, USA
[2]National Telecommunications Institute, Cairo 11786, Egypt
[3]Green Computing Systems Research Organization, Waseda University, Tokyo 169-8050, Japan

Corresponding author: Ahmed Sherif (ahmed.sherif@usm.edu)

ABSTRACT Nowadays, sensors, communications connections, and more powerful computing capabilities are added to automobiles, making them more intelligent. The primary goal was to eliminate the need for human control, making them Autonomous Vehicles (AVs). Consequently, researchers thought to put all that newly added computational power to use for other endeavors. Hence, Autonomous Vehicular Cloud Computing (AVCC) models were introduced. Nevertheless, this goal is not an easy undertaking, the dynamic nature of autonomous vehicles introduces a critical challenge in the development of such a distributed computing platform. Furthermore, it presents far complicated issues as far as security and protection of services associated with this framework. In this paper, we center around securing programs running on AVCC. Here, we focus on timing side-channel attacks which aim to leak information about running code, which can be utilized to reverse engineer the program itself. We propose to mitigate these attacks via obfuscated compilation. In particular, we change the control flow of an input program at the compiler level, thereby changing the program's apparent behavior and accompanying physical manifestations to hinder these attacks. We improve our previous ARM-based implementation to address its limitations and provide more comprehensive coverage for different programs. Our solution is software-based and generically portable - fitting different hardware platforms and numerous input program languages at the source level. Our findings prove a considerable improvement over our previous technique, which may provide more defense against timing side-channels.

INDEX TERMS Autonomous vehicles, autonomous vehicular cloud computing, software security.

I. INTRODUCTION

The computing power of autonomous vehicles (AVs) is quickly increasing. AVs are outfitted with different processing, memory and storage facilities, as well as computer vision technologies. In addition, there is a myriad of sensors and actuators that are connected to each other and to their surroundings via various communications interfaces, allowing them to drive themselves autonomously without the need for human control. Furthermore, AVs frequently collaborate to collect data from their surroundings and transmit it to distant servers, where it may be processed and analyzed to deliver various services.

The associate editor coordinating the review of this manuscript and approving it for publication was Tariq Umer.

In addition to navigation, some collaborative applications were introduced to monitor the environment and pollution levels or aid in wide-scale traffic management systems. Hence, a paradigm called the Internet of Vehicles (IoV) [2] was born, which is the next level of Wireless Sensor Network, where the cars themselves act as the information hub. Researchers took the computing potential found in AVs to the next level. They aim to utilize these smart cars' occasionally inactive computing capabilities to provide computing services as a utility. This model is called autonomous vehicles cloud computing (AVCC) [3].

Cloud computing is a relatively new technology that is currently game-changing in the industry. Users don't need to own powerful computing capabilities at their hands. Instead, they can rent as much power as they need in

a pay-as-you-go model. The advancement in communication technologies such as LTE and 5G allows a gradually ubiquitous spreading of this new paradigm. Cloud computing is being offered in different delivery models suiting different user needs. There is Software as a service, Platform as a service, and Infrastructure as a service. These models have something in common; some computation task is done remotely in a physically out of reach platform that the user cannot control or govern.

Remote code execution is a trending requirement in numerous usage scenarios, such as a case when a user is using a smartphone or a small computer. In other situations, some companies opted to use cloud computing platforms to allow their employees a more flexible working style. In times of hardship like nowadays, the pandemic forced many people to work from their homes, and they needed to access the company's computing resources seamlessly with the same functionalities. On the other hand, all of these usage scenarios suffer from common security threats and privacy concerns. More importantly, remote code execution on shared platforms that are physically inaccessible is inherently risky in terms of trustworthiness. That is to be confidential, integral, and available at time of need.

These requirements become more challenging in the field of AVCC. Although AVCC is essentially a cloud computing platform, using cars instead of stationary computers residing in some company's building introduced more challenging problems. The first obvious problem is that these cars are moving, which means that the communication interfaces will continually change the cloud formation. Although the organizational problems are addressed from an architectural perspective, they open the system to unknown threats every time a car enters or leaves the cloud [4]. Secondly, AVs are powered by embedded systems, which means they have power limitations and are also limited in processing capabilities, storage, and memory.

AVs are indeed a bonanza of a variety of security attacks such as Denial of Service (DoS), jamming, hijacking authentication, racketeering, copyright infringements, stealing data, sabotage,, and our current focus here, information leakage via side-channels and reverse engineering.

This new breed of attacks is resilient to traditional open security methods, relying on conventional cryptographic approaches. Digital signatures, certificates, and trusted platforms are examples of traditional cryptographic approaches that may not be sufficient. Since the code is running remotely, attackers may be able to view the actual decryption process and get the information [5]. Homomorphic encryption [6] is a fairly new technique to support encrypted execution of instructions. Even so, it is not quite applicable yet, since it requires complicated setup hardware and tremendous practical cost, which may not be suitable in the embedded systems environment. Therefore, we need a new practical security approach to remotely protect code execution from potential attackers who share the same physical hardware in the AVCC.

Here we propose to use a technique called obfuscation to hide the actual behavior of a running program by increasing its logical complexity [7]. The obfuscation makes the code difficult to understand by attackers; hence they cannot leak information about it. That's why this closed technique is called security by obscurity or security by design. In particular, we disrupt the normal control-flow of a running program, by introducing a compiler-based branch instruction transformation algorithm, which is applied dynamically and randomly to the input program. Therefore, we complicate the behavior of the control flow of the program, which will be reflected in its running time and other physical manifestations such as power consumption, electromagnetic and sound emissions. A side-channel attacker would need consistency in these manifestation to infer correlations about a running program and hence leak information about the running program's behaviour. By using our technique we would disrupt these correlations and hence thwart side-channel attacks.

This work presents an improvement to our obfuscation via compilation technique [7]. In particular, the contributions of this paper are:

- *first*, we enhance our obfuscation mechanism to protect programs running on AVCC platforms against information leakage via side-channel attacks which use timing analysis;
- *second*, we address the limitations of our initial implementation in some AVCC applications, specifically in the cases where there was a small number of opportunities for branch conversions, hence it limited control-flow obfuscation transformations.;
- *third,* we kept the advantages of our compiler-based software system, that's being input language agnostic and platform independent, which makes our system generic and easily applicable in AVCC platforms;
- *finally*, present an analytical study for our system and their effectiveness with regard to different input programs.

The remainder of this paper is organized as follows. Section II lays forth some of the fundamental notions that underpin the ideas we used in our suggested method and provides a summary of the literature relevant to our study, as well as their efficacy in contrast to our suggested approach. In section III, we discuss our network and threat models as they apply to our system. In part IV, we talk about how we're going to put our system in place. Section V depicts the experiments and the analysis of the data. Finally, in section VI, we wrap up the study and make some recommendations for further research.

II. BACKGROUND INFORMATION AND RELATED WORK

Before moving forward with laying out our proposed system, we have to build a firm background about the information that we used as well as a knowledge base of the related literature and how it connects to our work.

A. PRELIMINARIES

In this section we delve into more details about the most important constructs that we used in our system.

1) THE LLVM COMPILER

A compiler is a tool that converts high-level language programs into machine-level instructions. Compilers learn a lot of program's semantic information, which allows them to optimize output programs' performance.

LLVM [8] is a robust, open-source compilation infrastructure for building compilers. Historically, LLVM began as a research project known as (Low-Level Virtual Machine) developed by Vikram Adve and Chris Lattner at the University of Illinois [8], [9] to provide a static/dynamic compiler applicable for an arbitrary wide range of programming languages. Now LLVM is the official compiler for Apple products, including MAC OS X and iOS. The compiler is based on the famous Static Single Assignment (SSA) form [9] significantly simplifies developing compiler optimizations.

Moreover, LLVM is an extensible compiler suite that supports customization through modules and plugins. It allows developers to create plugins to customize the functionality of the compiler. For instance, a custom LLVM pass exists to implement the Control Flow Guard (CFG) exploit mitigation and things like binary instrumentation for fuzzing/code coverage purposes. The LLVM compiler suite, functions as a backend portion of a compiler that handles machine code generation from the LLVM IR (Intermediate Representation).

Compiler suites such as the Clang C/C++ compiler and other programming languages like the Swift and Rust compilers use the LLVM project as a backend. These compilers output LLVM IR code, which is then passed to LLVM to generate compiled binaries from the LLVM IR. Any compiler targeting LLVM as a backend automatically supports code generation for any architecture supported by LLVM, such as Intel X86 or ARM. The LLVM project also includes a linker (LLD) and other valuable utilities when developing compilers.

Due to its wide adoption and modular extensible architecture, LLVM is an excellent choice when writing plugins for code obfuscation purposes. By performing obfuscation at the LLVM IR level, it is possible to develop compiler passes for code obfuscation purposes that support multiple programming languages and instruction set architectures.

2) CONDITIONAL BRANCHES

Control-flow optimizations, such as branch optimizations, relax the control dependences in the program in order to facilitate parallelization and pipelining. Conditional branches are the main instructions that affect control-flow, as their outcome is generally not known until the runtime. Typically, these branches add a significant cost to the runtime of a program. As a result, conditional branches impede parallelism or pipelining attempts [10].

Historically, numerous scholars experimented with various ways to convert or resolve branches to increase program speed [11]. As a result, speculation was developed to reduce the impact of such control dependency while maintaining data correctness. Branch prediction is used in speculative execution to figure out the location of the next instruction beforehand. Then we can fetch, decode, and execute the next instruction as though the branch forecasts were consistently right [12]. The entire pipeline is flushed if a mistake is found during execution, the proper instruction is retrieved, and all operations completed are thrown away. By using predicates, which are a form of guarded instructions, the authors in [13] attempted to relieve such control dependences by using guard instructions called predicates. An instruction's execution is determined by the evaluation result of some guard expression.

Conditional branches can be classified into the following branch types [14]:

Forward Branch Changes the control-flow to a target after the branch but, generally, in the same loop nesting level.

Backward Branch Changes the control-flow to a target before the branch but, generally, in the same loop nesting level. A backward branch can be thought of as a loop; thus, the loop optimizations can be applied to it.

Exit Branch Transfers the control out of the loop nest, which terminates one or more loops.

Consequently, there are various techniques involved in optimizing branches within a piece of code [15]. They are listed below:

1) Straightening: put the target code instead of the branch, replaces some branches with the target code to get larger basic blocks.
2) Tail merging: unify tail branches of basic blocks to a single branch, which replaces identical tails with one tail and branch from the others.
3) Branch-to-Branch optimization: replaces a branch by a simpler one, which usually occurs when a branch target is another branch.
4) If simplification: eliminating the empty or constant-valued condition arms of an if-construct; it also deletes the empty if-constructs that may be found in the automatically generated or optimized code.
5) If-conversion: converts conditional branches into predicated instructions, supported by the underlying processor architecture.

In section IV, we discuss in details if-conversion optimizations. We utilized these branch optimization techniques for security through obscurity by varying their time cost to thwart probabilistic analysis of code execution.

B. RELATED WORK

Almost all of the concepts we used in our system have been studied before in some form or another; the originality is in combining the appropriate components in the proper order to better fulfill our security needs without making the system

too complicated. Some of the relevant work is discussed in this section.

1) CODE OBFUSCATION

First of all, we have to state that obfuscation has been around for some time [16]. It was introduced to manage the privacy of sensitive data in cloud computing platforms [17] and for program protection as in [18]. In [19], the authors present a study of Obfuscation, and deobfuscation tools in Android platforms. They investigated automated tools such as R8, ReDex, Obfuscapk, and DeGuard. Additionally, [20], [21] also investigated some other techniques to obfuscate android apps. Some of the most famous techniques are: Debug Information Removal, Function Call Indirection, Goto Instruction Insertion, Reordering, Arithmetic Branch Insertion, Nop Insertion, and Medhods Overload. Also, there several commercial Java obfuscators such as Zelix Klassmaster, Stringer, Allatori, DashO, DexGuard, ClassGuard, and Smoke.

However, the more prevalent use of obfuscation is in hiding malware and other foist software to evade scanning or analysis. An example work of such context is of [22]. They proposed a technique for obfuscating trigger-based malware code based on some conditions at the static compiler level. This scheme allows for evading malware analysis tools. They used the LLVM compiler to transform the input program into an obfuscated binary. The system captures a conditional input trigger that starts the malware; it derives an encryption key from the input, encrypts the code, and removes the key from the generated code. Thus analyzer programs cannot easily detect the start or execution of malware code. This system generates static obfuscated code essentially for malware triggering.

On the other hand, there has been extensive research as well on deobfuscation and automated analysis tools [23]–[25]. For example, in [26], the authors present a technique for extracting an executable program from the packed and obfuscated binary code. They developed a hardware-assisted software for import tables reconstruction and rebuilding obfuscated API names.

That said, our proposed obfuscation system generates dynamically, and every changing obfuscated binaries, which make our is generic in that sense ad suitable for application in the realm of AVCC platforms.

2) CLOUD SECURITY

There have been many attempts in the literature to solve the security problem in the remote execution platforms in general such as the cloud computing. Many recent papers surveyed the latest state of art work in that endeavor, such as [27]–[30]. However, to the best of our knowledge, none of them has considered using dynamic compilation technology to secure remote code execution. Here, we shed some light on some of these attempts.

a: TWIN CLOUDS

[31] propose securing the cloud by utilizing two clouds: a trusted private cloud (where the cloud is under the user's control) and a commodity public cloud. The private cloud is used for encrypting critical data and algorithms (setup phase). The commodity cloud is used for computing time-critical computations (trusted cloud queries) in parallel under encryption (the query phase). A user first sends his/her request to the trusted private cloud, which authenticates and encrypts the algorithm/data using a trust mechanism that is based on Yao's garbled circuits [32]. This process is based on two-party encryption to realize what is called verifiable computing [33]. That is computing the value of a function with minimal knowledge from participating parties. The system exposes the twin cloud architecture to programmers, increasing the cost of the software. Moreover, it incurs the extra cost of garbled circuit execution and communication between the clouds. Though, this work presented a practically efficient approach for secure computations as opposed to Fully Homomorphic Encryption (FHE), which aims to allow calculations on encrypted data without using additional helper information [34], [35].

b: HYPERVISOR SECURITY

[36] discussed the issue of how to trust a hypervisor. They present root trust static and dynamic management concepts. They suggest having a third-party certificate authority that provides certificates that can be used for remote attestation of a given platform; by extending the Trusted Computing Base (TCB) as per the Orange Book [37]. The difference between Static Root Trust Management (SRTM) and Dynamic Root Trust Management (DRTM) is that the latter can start a program in an Isolated Execution Environment (IEE) at any time, not just at boot time, which is a new root for trust chained from the initial state of the machine (a clean CPU state). Hence, a client can be assured that its virtual machine is integral since it has started from a trustworthy state and has not been modified or replaced by a malicious one. The system incurs a costly start overhead due to the chained trust mechanism. Moreover, the system is still susceptible to side-channels attacks from other virtual machines. Also, a downside of the system is that the technique relies only on verifying that the hash belongs to a list of trusted hashes, but that does not necessarily guarantee that it represents a trustworthy module. The certificate authority can be deceived by a fraudulent certificate issued by a malicious insider since the system relies only on a key for security in the launch process. Moreover, after the launch process, there is no way to guarantee the integrity or privacy of our computations on the cloud during runtime. There is also the risk of sabotage attacks via buffer and memory overflow exploits.

c: SECURE VIRTUAL ARCHITECTURE (SVA)

[38] present a new compiler-based virtual instruction set for executing code on a given system, including kernel and application code. The architecture provides instructions for object-level memory safety, control-flow integrity, and type safety, allowing it to monitor all privileged operations and control physical resources. They also provide custom instructions

to control memory layouts, such as allocation and explicit de-allocation instructions. Thus, this work only protects the system from sabotage attacks such as memory or buffer overflow attacks on a physical resource. However, the system is still susceptible to eavesdropping attacks, especially at the OS level. Moreover, the SVA sandboxing mechanism focuses only on the instruction set beyond the Intermediate Representation (IR) level and a code-generation phase.

That said, there are hardware-based commercial solution offered by major vendors in the cloud market. For example there is Intel TXT [39], ARM TrustZone [40], AMD SEV [41], and Intel SGX [42] technologies. The researchers in [43] studied these technologies and compared them with each other. However, the found that each technology can offer certain security guarantees that the other technologies do not provide under some particular settings. Therefore, it is up to IT security managers to choose which hardware to adopt according to their needs.

On the other hand, our work is a protection mechanism that is totally software based, which is easier and less expensive to set-up and more applicable to a wide range of usage scenarios. Moreover, it's independent of input programming language and architecture agnostic. That's because we focus on the intermediate representation level. This makes our proposed mechanism a suitable for a heterogeneous platform such as the AVCC.

3) SIDE CHANNELS

Side Channel Attacks (SCA) have been posing a great threat against different platforms and architectures. That's why there has been a great deal of research on how to launch them and how to thwart them equally such as [44]–[46].

For example, the authors in [47] present a software-based side channel attack that monitors power consumption statistics. They exploited the Intel Running Average Power Limit (RAPL) interface to gain unprivileged access, which allowed them to correlate and infer which instructions being executed and distinguish hamming weights of operands and memory access operation. This breach allowed them to leak information about the control flow of running program and to leak data and cryptographic keys as well. They also presented some non-trivial mitigation techniques to this attack.

In addition, the authors in [48], examined the vulnerabilities of ARM processor's instruction set architecture (ISA) and their susceptibility to SCAs. They also, surveyed different countermeasures to thwart these attacks. However, most of these attacks targeted cryptographic algorithms in order to leak information about encryption keys, but here, we aim to protect the execution trace of the running program using obfuscation. Thereby, we mangle with the correlation between a program's runtime behavior and the input data. Hence, we make it difficult for attackers to leak information about the program and/or reverse engineer it.

On the other hand, the researchers in [49] introduced a software tool that can generate a polymorphic version of a function of a program. They used different techniques like instruction reordering, changing addresses of registers, and dummy code insertion. Nonetheless, our techniques provides more diversification to the resulting technique since we employ a technique that randomly and dynamically changes the control-flow of the program and hence its execution time. Moreover, our scheme is simple to apply and already integrated within the compiler infrastructure which makes it a relatively light-weight technique. However, some of the aforementioned ideas, can be incorporated in our system to offer more protection. These, along with other similarly integrable methods will be investigated in future work.

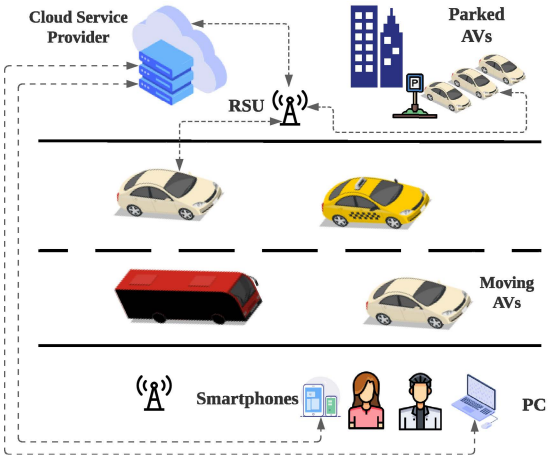

FIGURE 1. Autonomous vehicular cloud computing system.

III. SYSTEM DESCRIPTION

A. NETWORK MODEL

cloud computing has existed for a while now, hence the idea of utilizing AVs computing capabilities to create a similar on-demand computing platform coined Autonomous Vehicular cloud computing (AVCC). As illustrated in Figure 1, AVs communicate together and to the outside world using Road Side Units (RSUs) and many communication interfaces. Thereby, they can share their resources such as processing, storage, sensing and collaborate together to perform computation tasks which may need powerful processing. These AVs can be traveling along the road or parked in some place. Often, they may connect with remote servers to provide some service. The management and synchronization of such collaborative efforts and sharing of physical resources will be done by a designated AVCC controller, often one of the parked AVs, in order to balance the demand with the availability. The service provider will sell and manage the computing offerings through some kind of an interface with perspective clients who might be using their personal computer or even smartphones in any place around the world. A client should be able to execute a code or/and store data on some physically hardware selected by the AVCC controller securely and without any governance from the client side.

B. THREAT MODEL

Platforms that are utilize shared resources such as AVCC, often suffer from security attacks. That's because, attackers

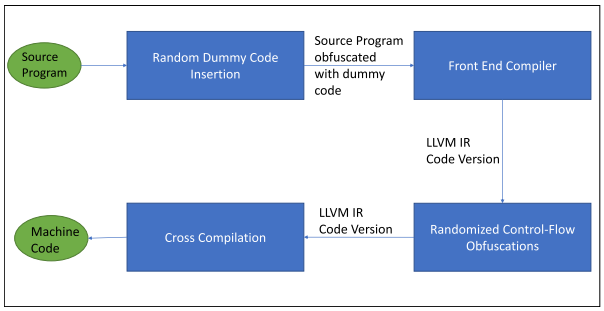

FIGURE 2. Flow chart explaining the steps of producing an obfuscated code version using our proposed system.

try to advantage of some loopholes to breach the system and obtain access to user data.

Here, we focus on non-invasive attacks that aim to leak information about running programs. These attacks are often easy to implement and quite scalabele and if executed on a large scale can threaten the entire platform. They are called side-channel attacks (SCA), which are based on monitoring of some physical phenomenon associated with the execution of a target program, such as it's power consumption, acoustic emission, resource access patters or running time. Then, the attacker might try to analyze this data in order to deduce some correlations about the behavior of the program being executed. Based on that, he might be able to learn which instructions were issued and back trace the execution of the program in order to reverse engineer it. His target might be stealing a copyrighted program or later tamper with it.

We assume that the attackers can be unauthorized intruders or other users of the cloud platform. We also assume that the service provider is honest but curious, meaning that they would not affect the integrity of our system and the correctness of the running program, but may try to leak information about the code in the same manner mentioned above. Either way, our proposed closed security-by-design technique should try to thwart side-channels from any adversary by obscurity and increasing logical complexity.

IV. PROPOSED TECHNIQUE

In this work, we extend our earlier systems that use obfuscated compilation, to work in the realm of AVCC [1]. In particular, We used the same idea and developed an enhancement to that system designed to better serve the ARM based architecture supporting the AVCC platform. In this section, we explain how our system was constructed how it works. We also addressed some architectural issues with the obfuscation transformations to suit the new platform.

ARM processors have a technology called predication. A predicate is a logical concept that adds a control flow decision point or an if-then-else functionality to the next group of instructions,thereby they are said to be conditional. In the original 32-bit architecture, there was a combination of four conditional codes that controlled 13 predicated instruction. Later in 1994 the Thumb instruction set was introduced and the inventors sought to eliminate this sort of conditional execution in order to reduce the size of instruction for a size of 16 bits. Then again in the modified version coined Thumb-2, they overcome the size problem by introducing special instruction which only provides predicates to the following instructions.

In our system, we used a similar concept but with a dynamically changing *guard* bit that controls the branch conversion process based on the bit's random value (either True or False), which is the basis of our control-flow obfuscation technique. We aimed to integrate this process within the LLVM infrastructure, without affecting its integrity or the correctness of the compiled program. Therefore, we modified LLVM compiler framework to implement a dynamic obfuscation scheme using randomized transformation of conditional branches. When a source program is fed into the compilation framework, our system first examines each code block (typically a function) looking for these branch conversion opportunities. Then, these convertible branches are flagged by a bit as a sort of a *predicate* to decide whether to transform that branch or not. This flag bit can be dynamically changed with each compilation phase in order to change the overall control flow of the program. Moreover, each flag can be modified independently which allows us to add more disruption to the input program at certain areas of the code (say hotspot function) which in turn will manifest in significantly added unpredictability of the timing behaviour of that program. This would allow us to create sort of diversified code versions for the same input program, having different control flow behaviour, while maintaining its correctness and functionality. Thereby, making it quite complicated for attackers launching timing SCA to build correlations about the behaviour of the running program, hence he will not be able to leak information about it.

Figure 2 shows the steps of creating diversified machine code versions from an input source program. These code version are tailored to the ARM platform target using LLVM's cross-compiler. Figure 3 shows how we ported our LLVM based system to suit the embedded system architecture supporting the AVCC platform. The figure detail how the cross-compilation process works by utilizing different static and shared libraries and toolchains according to the target object. Programs running on ARM-based systems can be one of two things, bare-metal infrastructures and OS controlled platforms. In the bare-metal model, you would need to compile all program libraries and every associated frameworks or toolchains and link them together in the resulting executable machine code. While in the latter model, often a Linux based OS (such as Linaor OS) is the one governing the system. Therefore, you wouldn't need to compile and link every little piece of code needed in your program. But on the downside, the ARM processor itself need to be powerful enough to support the OS and other programs to be executed on it. All of these conditions should be kept in mind, since they would greatly affect the resulting code size and hence its

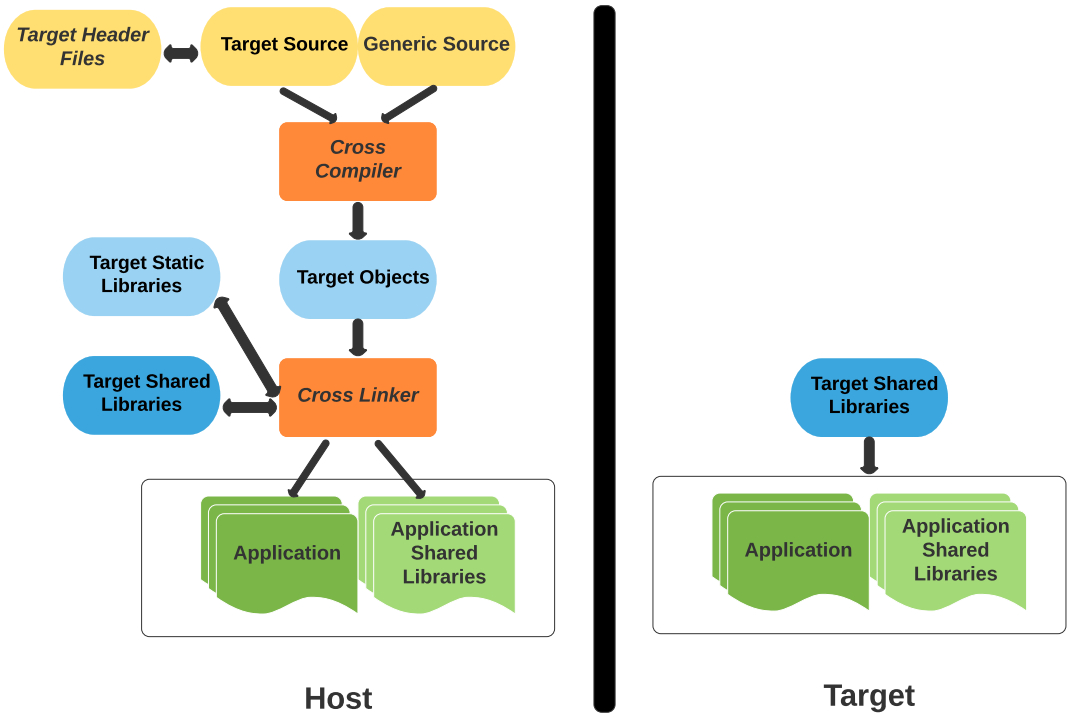

FIGURE 3. LLVM cross compilation steps.

execution time. Nevertheless, in both scenarios, our obfuscation scheme will work essentially in the same manner.

That said, during the experiments performed in [1], we noticed that some a limited number of branch conversion opportunities due to its underlying control-flow structure. It means that the number of conditional branches converted without threatening these conditions expectedly varies from one program to another. That's why, in the current implementation, we introduced an LLVM plugin (Transformation) to randomly insert junk code into the input program. This inserted code could be just random instructions or an entire dummy program running within the actual input program without compromising its functionality. The purpose of this junk code is to maximize the branch conversion opportunities in the input program by introducing new lines of code, which will further hinder reverse engineering and analysis of the compiled code.

In particular, the junk code insertion is a ModulePass, meaning that the pass gets invoked on every module (source code file) during compilation. However, LLVM supports other modules such as FunctionPass that runs on every function, and BasicBlockPass that runs on every basic block within the program, which can be further leveraged in future work.

Note that this pass begins by creating a global variable that is referenced by the inserted junk code. This is because some LLVM optimizations attempt to remove dead code from the compiled program for optimization purposes. This optimization also has the unfortunate side effect of removing the junk code that we have inserted to evade signature-based detection. If the junk code references a global variable, it is not marked as dead code and deleted.

After creating the global variable, the pass uses a loop to iterate through each function, its corresponding basic blocks, and each of the instructions within those basic blocks. It then chooses and inserts random lines of code within the input program. These instructions should be simple enough to make sure not to overly complicate the input program or significantly affect its overall performance. Nevertheless, the manifestation of these modification would be noticed in the runtime results, which serves our goal in mitigating side-channel attacks.

TABLE 1. Comparison of range of normalized runtime differences in both versions of our system.

Benchmark Name	Old System	New System
Float	-0.3 : 2 %	2.4 : 4.9 %
IntMM	-16 : 5 %	24 : 49 %
Perm	0.95 : 3.9 %	2 : 10 %
Queen	-5.8: 3.8 %	3.6 : 10.8 %
Quick Sort	-4 : 3.2 %	3.8 : 9.9 %
Puzz	0.7 : 3.9 %	9.4 : 12.6 %
Oscar	-1.8 : 4.7 %	1.5 : 6.6 %

V. EXPERIMENTS AND RESULTS

We set up our environment as follows. An HP Notebook model 15-DY1023DX acting as the host machine running WSL2 Ubuntu 20.04.1 LTS x86_64 version, and the target

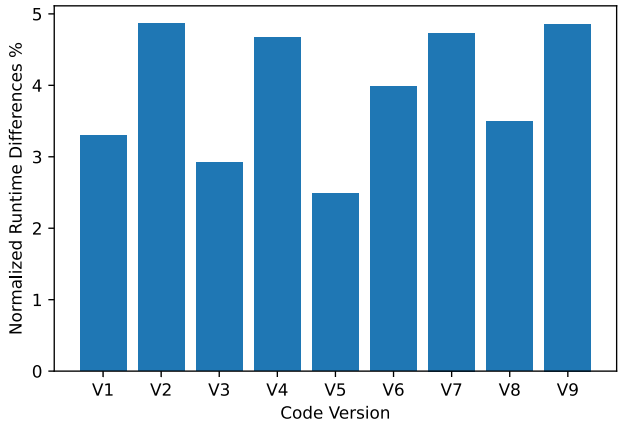

FIGURE 4. Obfuscated code versions of the float program and their corresponding percentage of normalized runtime differences with respect to the original unmodified version.

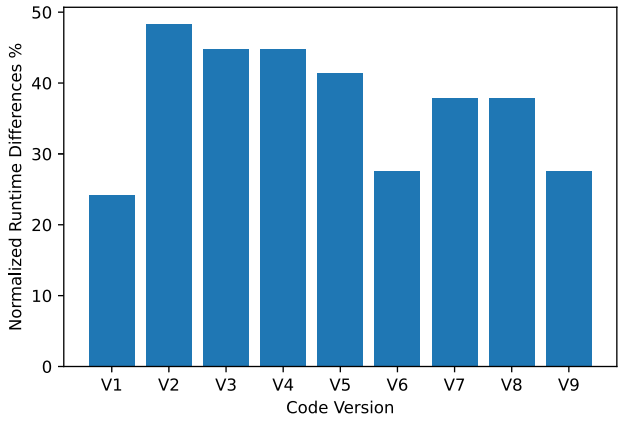

FIGURE 5. Obfuscated code versions of the IntMM program and their corresponding percentage of normalized runtime differences with respect to the original unmodified version.

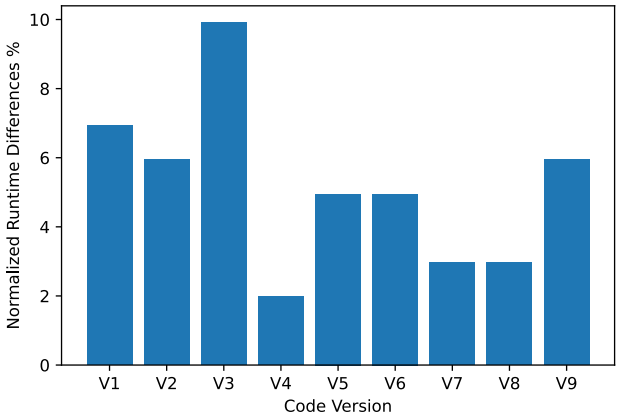

FIGURE 6. Obfuscated code versions of the perm program and their corresponding percentage of normalized runtime differences with respect to the original unmodified version.

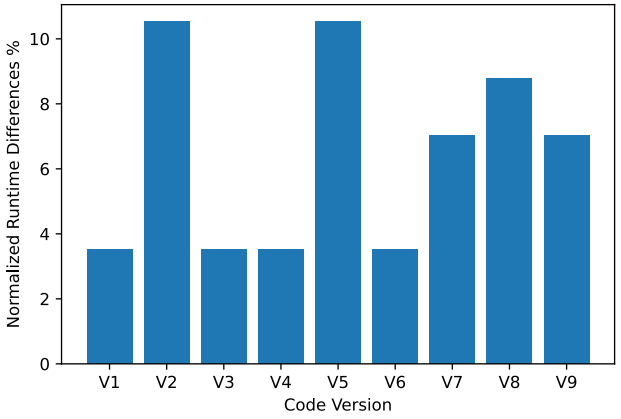

FIGURE 7. Obfuscated code versions of the queen program and their corresponding percentage of normalized runtime differences with respect to the original unmodified version.

machine was a Linux-based 32bit ARM platform. For the sake of proving our system's validity, we simulated the target platform using an ARM Emulator (QEMU) [50]. This software simulates an Thumb-2 Arm microprocessor along with its system-level architecture. Nevertheless, we plan to test the system with real hardware kits in our future work.

We added our obfuscation extension to LLVM framework version 12, and used it to cross-compile the source of the input programs and produce machine code targeting the ARM platform. Our technique is easily integratable with any other version of the LLVM compile that supports the "If Conversion" transformation.

As a proof of concept, we choose some simple yet standard benchmark programs as the subject for our experiments. These benchmarks are well-known algorithms borrowed from the test set of the Gem5 software [51]. We fed these programs to our system, which first added some dummy code that contains branch instructions. This dummy code is randomly selected from a pool of code base programs. We did that to avoid using the same code multiple time, hence this could present a vulnerability in our system. Then the modified code would be inserted into the control-flow obfuscation pass. Which in turn does random If conversion transformation to the program. Finally, a machine code tailored for the ARM platform is produced using the LLVM cross compiler, as explained before in Figure 3. The previous steps were summarized in Figure 2.

We did the same process multiple times for each input program, in order to produce different code version for the same program. Then, we compared the running time of each code version. This study of time measurement is the basis for thwarting timing side-channel attack because the greater and the more unpredictable the runtime, the more it will be difficult for an attacker to make statistical correlations about the behavior of the program, hence he can't leak information about it or reverse engineer it.

Figures 4 through 10 show the normalized runtime difference percentages across the various versions that we produced for each individual benchmark. Each of these different program versions corresponds to a different bitmask, hence another predicated instruction series and consequently different running times, although the programs have exactly the

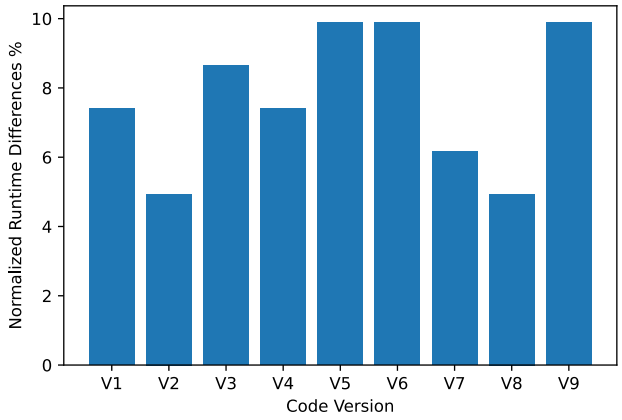

FIGURE 8. Obfuscated code versions of the quick sort program and their corresponding percentage of normalized runtime differences with respect to the original unmodified version.

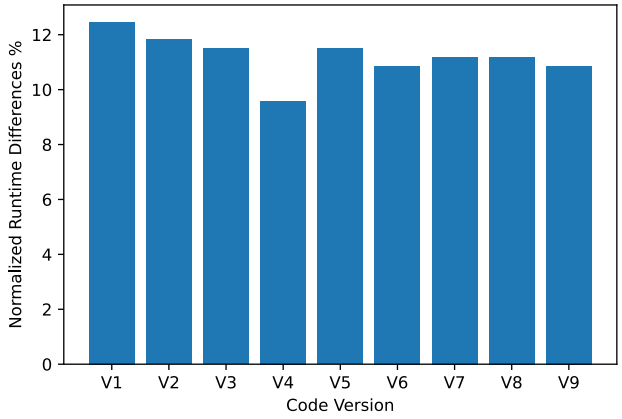

FIGURE 9. Obfuscated code versions of the puzz program and their corresponding percentage of normalized runtime differences with respect to the original unmodified Version.

same functionality. The running times were normalized in comparison to the runtime of the original unmodified code version, and the results were plotted in these figures respectively. The equation for the normalized runtime difference can be expressed as follows:

$$\Delta T\% = ((T_{Vi} - T_O)/T_O) \times 100 \tag{1}$$

where T_{Vi} is the runtime for that specific obfuscated code version (which in turn corresponds to s specific bitmask), T_O is the runtime for the original unmodified version. This measure service as both a performance metric and indicator of the robustness of our security proposal.

As we can see that these different code version all have one thing in common, they were expectedly slower than the original code version, due to our current system enhancement that is the insertion of the randomly selected dummy code with more branch opportunities. It also went under the same control-flow obfuscation transformation and produced varying code versions with different runtimes.

To prove on how this enhancement affected the overall system, we experimented on the same benchmarks using

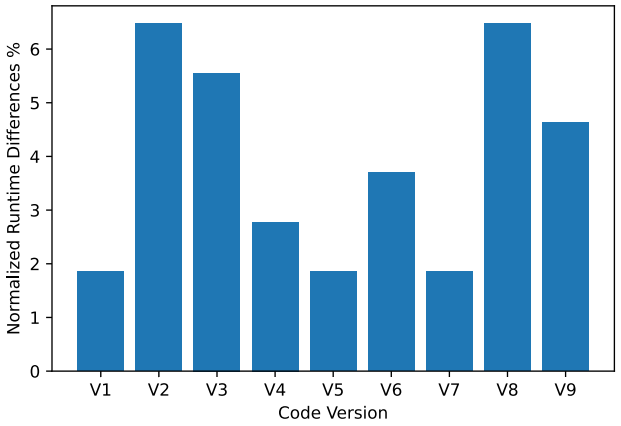

FIGURE 10. Obfuscated code versions of the oscar program and their corresponding percentage of normalized runtime differences with respect to the original unmodified version.

the old and the new techniques on the same platform and collected the overall range of normalized runtime differences. We summarized the comparison of both techniques in Table 1.

For example in Figure 4 we can see that these normalized differences varied between 2.4% and 4.9%, which means they ran a little slower than the original code, with different percentages for each code version. On the other hand from Table 1, we know that this range was −0.3% to 2%, which was quite small, proving that we needed this sort of enhancement.

The same goes for the IntMM benchmark plotted in Figure 5, showing a range of 24% to 49% as compare to the −16% to 5% with the old technique as per Table 1. Also the Perm program in Figure 6 had a range of 2% to 10% as opposed to 0.95% to 3.9%. In addition, the Queen program shown in Figure 7 showed similar results, ranging between 3.6% to 10.8% as opposed to −4% to 3.2%. As for the Quick Sort program, which we know from our previous work [1], has many recursive calls, thus it was more challenging to perform the if-conversion transformation. Having the opportunities for predication are limited [52], the time changes were also minor −4% to 3.2% as per Table 1. But with our new technique we achieved a range of 3.8% to 9.9%, and the statistics from its code versions were plotted in Figure 8.

On the other hand for the Puzz program shown in Figure 9, although the range is between 9.4% to 12% which is comparably larger than 0.7% to 3.9%, we have to note the width of the newly introduced range (the difference between the maximum and minimum values) is somewhat smaller than some of previous cases, hence the difference between the code versions themselves was not as big as it was in other programs. This could have happened because the inserted random dummy code didn't have many branch conversion opportunities, hence we need a more concise way to choose dummy code instead of just randomly inserting it. This needs to be addressed in future work, because as more as we have significant time changes, we can disrupt timing side-channel attacks.

Also, for the Oscar program, shown in Figure 10, the old technique achieved a range of −1.8% to 4.7%, and now with the new technique we have a range between 1.5% to 6.6% which some would say a smaller performance gain. Also, they could have happened because of a poor choice for the random dummy code. Though, unlike the Puzz program, here the difference between the code versions is quite noticeable, meaning that the results could be acceptable in mitigating the attacks.

In summary, we can see from these different results that in most programs, we achieved considerable enhancement over the old technique as per the resulting normalized time differences. That said, in some cases we need some way to efficiently insert random dummy code that would produce more runtime changes. We intend to further investigate that in our future work.

VI. CONCLUSION

AVCC platforms are fairly new and they are still under investigation. Security and privacy problems are among the most important issues facing there wide-scale adoption. One of the most serious security threats are side channel attacks (SCA). In our work, we proposed an obfuscation mechanism to protect software running on AVCC against timing SCA. Virus and malware developers use obfuscation to hide their code from scanners and detectors. We used similar concepts to implement a dynamic yet randomized control-flow obfuscation technique using conversion of conditional branches. This would result in seemingly unpredictable disruptions to normal code behaviour, making it difficult for attackers to leak information about the running program. In this current approach we provide enhancements to our early work. In particular, we compensate for the cases where there were limited opportunities for conditional branch conversions, which previously introduced hindrance to our obfuscation mechanisms. Therefore, we were able to extend more protection for a wider range of generic programs running on embedded system based platforms, which is typically the case in hand in AVCC.

The results show that our LLVM-based obfuscation system is platform agnostic, independent of the input language and quite lightweight, which makes it a good candidate for application the AVCC platform.

REFERENCES

[1] M. Hataba, A. Sherif, and R. Elkhouly, "A proposed software protection mechanism for autonomous vehicular cloud computing," in *Proc. IEEE Int. Midwest Symp. Circuits Syst. (MWSCAS)*, Aug. 2021, pp. 878–881.

[2] H. Zhou, W. Xu, J. Chen, and W. Wang, "Evolutionary V2X technologies toward the internet of vehicles: Challenges and opportunities," *Proc. IEEE*, vol. 108, no. 2, pp. 308–323, Feb. 2020.

[3] R. W. L. Coutinho and A. Boukerche, "Guidelines for the design of vehicular cloud infrastructures for connected autonomous vehicles," *IEEE Wireless Commun.*, vol. 26, no. 4, pp. 6–11, Aug. 2019.

[4] J. Cui, L. S. Liew, G. Sabaliauskaite, and F. Zhou, "A review on safety failures, security attacks, and available countermeasures for autonomous vehicles," *Ad Hoc Netw.*, vol. 90, Jul. 2019, Art. no. 101823.

[5] C. P. García, S. ul Hassan, N. Tuveri, I. Gridin, A. C. Aldaya, and B. B. Brumley, "Certified side channels," in *Proc. 29th USENIX Secur. Symp. (USENIX Secur.)*, 2020, pp. 2021–2038.

[6] C. Gentry and D. Boneh, *A Fully Homomorphic Encryption Scheme*, vol. 20, no. 9. Stanford, CA, USA: Stanford Univ., 2009.

[7] M. Hataba and A. El-Mahdy, "Cloud protection by obfuscation: Techniques and metrics," in *Proc. 7th Int. Conf. P2P, Parallel, Grid, Cloud Internet Comput.*, Nov. 2012, pp. 369–372.

[8] *The LLVM Compiler Infrastructure*. Accessed: Feb. 1, 2022. [Online]. Available: http://www.llvm.org/

[9] C. A. Lattner, "LLVM: An infrastructure for multi-stage optimization," M.S. thesis, Dept. Comput. Sci., Univ. Illinois, Champaign, IL, USA, 2002.

[10] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proc. 10th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang. (POPL)*, 1983, pp. 177–189, doi: 10.1145/567067.567085.

[11] J. E. Smith, "A study of branch prediction strategies," in *Proc. 25 Years Int. Symposia Comput. Archit. (Selected Papers) (ISCA)*, 1998, pp. 135–148. [Online]. Available: http://dl.acm.org/citation.cfm?id=800052.801871

[12] J. L. Hennessy and D. A. Patterson, *Comput. Architecture: A Quant. Approach*. Amsterdam, The Netherlands: Elsevier, 2011.

[13] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M.-W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 2, pp. 138–150, May 1995, doi: 10.1145/225830.225965.

[14] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. Patt, "Branch classification: A new mechanism for improving branch predictor performance," *Int. J. Parallel Program.*, vol. 24, no. 2, pp. 133–158, Apr. 1996.

[15] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann, 2002.

[16] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Dept. Comput. Sci., Univ. Auckland, New Zealand, Tech. Rep. 148, 1997.

[17] M. Mowbray, S. Pearson, and Y. Shen, "Enhancing privacy in cloud computing via policy-based obfuscation," *J. Supercomput.*, vol. 61, no. 2, pp. 267–291, Aug. 2012.

[18] M. Hataba and A. El-Mahdy, "Cloud protection by obfuscation: Techniques and metrics," in *Proc. 7th Int. Conf. P2P, Parallel, Grid, Cloud Internet Comput.*, Nov. 2012, pp. 369–372.

[19] G. You, G. Kim, S.-J. Cho, and H. Han, "A comparative study on optimization, obfuscation, and deobfuscation tools in android," *J. Internet Serv. Inf. Secur.*, vol. 11, no. 1, pp. 2–15, 2021.

[20] J. Park, H. Kim, Y. Jeong, S. Cho, S. Han, and M. Park, "Effects of code obfuscation on Android app similarity analysis," *Proc. J. Wireless Mob. Netw. Ubiquitous Comput. Dependable Appl.*, vol. 6, no. 4, pp. 86–98, Dec. 2015.

[21] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding Android obfuscation techniques: A large-scale investigation in the wild," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.*, Cham, Switzerland: Springer, 2018, pp. 172–192.

[22] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2008, pp. 1–13.

[23] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 674–691.

[24] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "CoDisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 745–756.

[25] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: A semantics-based approach," in *Proc. 18th ACM Conf. Comput. Commun. Secur. (CCS)*, 2011, pp. 275–284.

[26] B. Cheng, J. Ming, E. A. Leal, H. Zhang, J. Fu, G. Peng, and J.-Y. Marion, "Obfuscation-resilient executable payload extraction from packed malware," in *Proc. 30th USENIX Secur. Symp. (USENIX Secur.)*, 2021, pp. 3451–3468.

[27] B. Alouffi, M. Hasnain, A. Alharbi, W. Alosaimi, H. Alyami, and M. Ayaz, "A systematic literature review on cloud computing security: Threats and mitigation strategies," *IEEE Access*, vol. 9, pp. 57792–57807, 2021.

[28] I. Kanwal, H. Shafi, S. Memon, and M. H. Shah, "Cloud computing security challenges: A review," in *Cybersecurity, Privacy and Freedom Protection in the Connected World* (Advanced Sciences and Technologies for Security Applications). Cham, Switzerland: Springer, 2021, pp. 459–469.

[29] M. K. Sasubilli and R. Venkateswarlu, "Cloud computing security challenges, threats and vulnerabilities," in *Proc. 6th Int. Conf. Inventive Comput. Technol. (ICICT)*, Jan. 2021, pp. 476–480.
[30] D. Sampson and M. M. Chowdhury, "The growing security concerns of cloud computing," in *Proc. IEEE Int. Conf. Electro Inf. Technol. (EIT)*, May 2021, pp. 050–055.
[31] S. Bugiel, S. Nürnberger, A.-R. Sadeghi, and T. Schneider, "Twin clouds: Secure cloud computing with low latency," in *Communications and Multimedia Security*, B. De Decker, J. Lapon, V. Naessens, and A. Uhl, Eds. Berlin, Germany: Springer, 2011, pp. 32–44.
[32] A. C.-C. Yao, "How to generate and exchange secrets," in *Proc. 27th Annu. Symp. Found. Comput. Sci. (SFCS)*, Oct. 1986, pp. 162–167, doi: 10.1109/SFCS.1986.25.
[33] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Advances in Cryptology—CRYPTO 2010*, T. Rabin, Ed. Berlin, Germany: Springer, 2010, pp. 465–482.
[34] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Advances in Cryptology—EUROCRYPT 2011*, K. G. Paterson, Ed. Berlin, Germany: Springer, 2011, pp. 129–148.
[35] N. P. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *Public Key Cryptography PKC 2010*, P. Q. Nguyen and D. Pointcheval, Eds. Berlin, Germany: Springer, 2010, pp. 420–443.
[36] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *Proc. IEEE Symp. Secur. Privacy*, May 2010, pp. 143–158.
[37] D. C. Latham, *Department of Defense Trusted Computer System Evaluation Criteria*. Richmond, VA, USA: Department Defense, United States of America, 1986.
[38] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 6, pp. 351–366, Oct. 2007, doi: 10.1145/1323293.1294295.
[39] W. Futral and J. Greene, *Intel Trusted Execution Technology for Server Platforms: A Guide to More Secure Datacenters*. Berlin, Germany: Springer Nature, 2013.
[40] S. Pinto and N. Santos, "Demystifying arm TrustZone: A comprehensive survey," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 1–36, Nov. 2019.
[41] "Strengthening VM isolation with integrity protection and more," Adv. Micro Devices, Santa Clara, CA, USA, White Paper, 2020.
[42] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.
[43] O. Demigha and R. Larguet, "Hardware-based solutions for trusted cloud computing," *Comput. Secur.*, vol. 103, Apr. 2021, Art. no. 102117.
[44] J. V. Cleemput, B. Coppens, and B. D. Sutter, "Compiler mitigations for time attacks on modern ×86 processors," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 1–20, Jan. 2012.
[45] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–18.
[46] C. Shen, C. Chen, and J. Zhang, "Micro-architectural cache side-channel attacks and countermeasures," in *Proc. 26th Asia South Pacific Design Autom. Conf.*, Jan. 2021, pp. 441–448.
[47] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based power side-channel attacks on ×86," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 355–371.
[48] D. McCann, K. Eder, and E. Oswald, "Characterising and comparing the energy consumption of side channel attack countermeasures and lightweight cryptography on embedded devices," in *Proc. Int. Workshop Secure Internet Things (SIoT)*, Sep. 2015, pp. 65–71.
[49] N. Belleville, D. Couroussé, K. Heydemann, and H.-P. Charles, "Automated software protection for the masses against side-channel attacks," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, pp. 1–27, Dec. 2018.
[50] *QEMU: The FAST! Processor Emulator*. Accessed: Feb. 1, 2022. [Online]. Available: https://www.qemu.org/
[51] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, and R. Sen, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.
[52] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, *Conversion of Control Dependence to Data Dependence*. New York, NY, USA: Association for Computing Machinery, 1983.

MUHAMMAD HATABA (Member, IEEE) received the B.Sc. degree in electronics engineering, majoring in computers and systems from the Faculty of Engineering, Mansoura University, Dakahlia, Egypt, in 2008, and the M.Sc. and Ph.D. degrees in computer science and engineering from the Egypt-Japan University of Science and Technology (E-JUST), Alexandria, Egypt, in 2013 and 2019, respectively. In 2015, he was a Visiting Research Fellow at the Professor Ueda Laboratory, Department of Computer Science and Engineering, School of Fundamental Science and Engineering, Waseda University, Tokyo, Japan. He is currently with the School of Computing Sciences and Computer Engineering, The University of Southern Mississippi, Hattiesburg, MS, USA. He is also an Assistant Professor at the Computers and Systems (CS) Department, National Telecommunication Institute (NTI), affiliated with the Ministry of Communications and Information Technology (MCIT), Cairo, Egypt. He has numerous publications in various aspects related to cybersecurity applications. He has also an extensive background in the fields of cloud computing, computer networks, compilers, and software engineering. His research theme was "Code Protection by Obfuscated Dynamic Compilation." His research interests include autonomous vehicles, deep learning, smart grids, and code protection.

AHMED SHERIF (Senior Member, IEEE) received the M.Sc. degree in computer science and engineering from the Egypt-Japan University of Science and Technology (E-JUST), in 2014, and the Ph.D. degree in electrical and computer engineering from Tennessee Tech University, Cookeville, TN, USA, in August 2017. He is currently an Assistant Professor with the School of Computing Sciences and Computer Engineering, The University of Southern Mississippi (USM), Hattiesburg, MS, USA. He is the author of numerous articles published in major IEEE conferences and journals, such as IEEE International Conference on Communications (IEEE ICC), IEEE Vehicular Technology Conference (IEEE VTC), IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY (IEEE TVT), and IEEE INTERNET OF THINGS JOURNAL. His research interests include cybersecurity, security and privacy-preserving schemes in autonomous vehicles (AVs), vehicular ad hoc networks (VANETs), the Internet of Things (IoT) applications, autonomous vehicle cloud computing (AVCC), and smart grid advanced metering infrastructure (AMI) network. He served as a Reviewer for several journals and conferences, such as IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY, IEEE INTERNET OF THINGS JOURNAL, and the journal of *Peer-to-Peer Networking and Applications*.

REEM ELKHOULY received the M.Sc. and Ph.D. degrees in computer science and engineering from the Egypt-Japan University of Science and Technology (E-JUST), Alexandria, Egypt, in 2012 and 2016, respectively. She is currently an Adjunct Researcher at the Green Computing Systems Center (GCS), Waseda University, Japan, and an Assistant Professor at the Computers and Automatic Control Engineering Department, Faculty of Engineering, Tanta University, Egypt. She has authored several papers presented in peer-reviewed conferences. She has published two manuscripts in *ACM Transactions on Architecture and Code Optimization* (TACO). Her research interests include compilers and code optimizations, computer architecture, high performance computing (HPC), neural networks (NNs), human–computer interaction (HCI), and agent based simulation (ABS).