IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 15, 2021, accepted March 5, 2021, date of publication March 29, 2021, date of current version April 8, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3069223

MeetGo: A Trusted Execution Environment for Remote Applications on FPGA

HYUNYOUNG OH[1,2], KEVIN NAM[1,2], SEONGIL JEON[1,2], YEONGPIL CHO[3], AND YUNHEUNG PAEK[1,2], (Member, IEEE)

[1]Department of Electrical and Computer Engineering, Seoul National University, Seoul 08826, Republic of Korea
[2]Inter-University Semiconductor Research Center (ISRC), Seoul National University, Seoul 08826, Republic of Korea
[3]Department of Computer Science, Hanyang University, Seoul 04763, Republic of Korea

Corresponding authors: Yeongpil Cho (ypcho@hanyang.ac.kr) and Yunheung Paek (ypaek@snu.ac.kr)

This work was supported in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) Grant funded by the Korean Government (MSIT) under Grant 2018-0-00230 (Development on Autonomous Trust Enhancement Technology of IoT Device and Study on Adaptive IoT Security Open Architecture based on Global Standardization [TrusThingz Project]) and Grant 2020-0-00325 (Traceability Assurance Technology Development for Full Lifecycle Data Safety of Cloud Edge) and Grant 2020-0-01840 (Analysis on Technique of Accessing and Acquiring User Data in Smartphone), in part by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIT) under Grant NRF-2020R1A2B5B03095204, in part by the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University, in 2021, and in part by the research fund of Hanyang University under Grant HY-2020. The EDA tool was supported by the IC Design Education Center (IDEC), South Korea.

ABSTRACT *Remote computing* has emerged as a trendy computing model that enables users to process an immense number of computations efficiently on the remote server where the necessary data and high-performance computing power are provisioned. Unfortunately, despite such an advantage, this computing model suffers from *insider threats* that are committed by adversarial administrators of remote servers who attempt to steal or corrupt users' private data. These security threats are somewhat innate to remote computing in that there is no means to control administrators' unlimited data access. In this paper, we present our novel hardware-centric solution, called *MeetGo*, to address the intrinsic threats to remote computing. MeetGo is a field-programmable gate array (FPGA)-based trusted execution environment (TEE) that aims to operate independently of the host system architecture. To exhibit the ability and effectiveness of MeetGo as a TEE ensuring secure remote computing, we have built two concrete applications: cryptocurrency wallet and GPGPU. MeetGo provides a trust anchor for these applications that enable their users to trade cryptocurrency or to run a GPGPU program server on a remote server while staying safe from threats by insiders. Our experimental results clearly demonstrate that MeetGo incurs only a negligible performance overhead to the applications.

INDEX TERMS Field-programmable gate array (FPGA), remote computing, remote attestation, secure communication channel, trusted execution environment (TEE).

I. INTRODUCTION

In *remote computing*, users transmit their own data to a remote application and receive its result. The significant benefit that users expect from remote computing is that they can perform a broad spectrum of computations—from small to large scale—directly on the remote site at lower costs and with better performance than on their own facilities. Cloud technology is one good example that serves this benefit from remote computing to cloud users. For example, users can entrust the cloud with personal information management (e.g., google-cloud), customer relation management (e.g., Salesforce), or a big data analysis based on machine learning (e.g., AWS-AI and Azure-AI).

With the growing popularity of remote computing services, security is becoming an ever-increasingly important issue for service providers and users. For instance, in one type of cloud, Software as a Service, everything, including applications and remote user data, is managed by the cloud server. It is known that data security compliance issues are somewhat intrinsic

The associate editor coordinating the review of this manuscript and approving it for publication was Alex Noel Joseph Raj.

to these types of cloud or remote computing mainly because the privacy and integrity of user data are built on the trust in the remote server. To be specific, as the server (or its administrator) normally has to manage all computing resources in the system, it is entrusted with the full privilege of controlling access to private data belonging to users. Unfortunately, any server dealing with private data entrusted by remote owners innately entails a security risk, called an *insider threat*, which comes from a privileged insider (i.e., administrator) of a server who turns into an adversary trying to steal or tamper with clients' data. According to [1], about a third of cyberattacks are suspected of being the result of insider threats.

A well-known solution to thwart such threats by insiders is building a *trusted execution environment* (TEE) [2] within a server for remote users. The TEE aims to ensure the privacy and integrity of user code and data loaded on the server. Applications loaded in the TEE are guaranteed to run and process data in an isolated environment securely from the rest of the host system, namely the *rich execution environment* (REE), administered by privileged insiders. Private user data are stored in secure storage shielded from the REE, and sensitive functions are executed inside a TEE without interference from the REE. Therefore, even if malicious insiders have full control over the REE, in principle, they cannot corrupt or leak remote user data processed inside a TEE.

Intel SGX provides a readily available TEE, called an enclave, that is isolated from all software entities including the kernel and the hypervisor. An enclaves is built up on top of an enclave page cache (EPC), encrypted memory regions so that any arbitrary accesses from outside are impossible. Therefore, SGX enclaves can provide remote users with reliable protection against insider threats. However, using SGX for remote computing poses a severe challenge in terms of scalability because the EPC is statically sized as 128 MB, which is too limited to execute data-intensive computations. Enclaves spending memory exceeding the size of the EPC bring about frequent memory swapping between the EPC and non-EPC, which results in considerable performance degradation. In addition, CPU where SGX enclaves are implemented is not optimized inherently for highly parallel workloads such as a machine learning based big data analysis. As such, SGX enclaves are somewhat unattractive to be leveraged on remote computing for a broad spectrum of workloads.

In this paper, hereby, we aim to realize another TEE that is solid in terms of both security and scalability, enabling remote computation that is robust against insider threats and efficient even on various performance requirements. Our approach to achieve the goal is to leverage an emerging architecture, called a CPU/FPGA hybrid architecture [3]–[5]. More specifically, we have built a TEE specialized for remote computing, called *MeetGo*, on a field-programmable gate array (FPGA). The foremost reason why FPGA is suitable for this purpose is that it is physically isolation from the would-be malicious CPU. Therefore, FPGA can serve as a TEE as long as proper control mechanisms are implemented to regulate arbitrary access to the inside of it. In fact, a recent study [6] has shown that FPGA can securely run a security-oriented application such as secure storage using this isolation feature. Another important reason is that FPGA is versatile because its internal hardware logic can be programmed dynamically. By deploying customized acceleration modules, FPGA can efficiently run a wide spectrum of applications such as data analytics, media processing, artificial intelligence, network security, finance, and genomics, as demonstrated in practice [7]–[12]. Even better, optimized logic design allows FPGA to achieve a higher power-efficiency than other computing hardware such as the CPU and the GPU.

To realize MeetGo on FPGA which is originally employed as a workhorse for the CPU in a CPU/FPGA hybrid architecture, we have implemented security mechanisms that are necessary for trustworthy remote computing. To be specific, we first have devised a remote attestation mechanism that can verify the integrity of the applications that exist as the form of hardware logic. We have also implemented an isolation mechanism to block unauthorized access to the applications from the malicious CPU. Lastly, we have developed a secure communication mechanism to allow secure transmissions of sensitive data between the installed applications and remote users. All of these are backed by a robust hardware trust anchor rather than relying on software implementations, as described in section III. One thing to note about MeetGo is that it can collaborate with other TEEs on the CPU side to improve their performance reliably. For example, an SGX enaclave can be connected to MeetGo as a remote user, and then it can entrust burdensome computations to MeetGo equipped with accelerator logics to process them more efficiently.

The purpose of this paper is to exhibit the ability and effectiveness of MeetGo as a TEE that ensures secure remote computing. For empirical demonstrations, we have built two trusted applications on MeetGo. One is a cryptocurrency wallet application described in section IV. The security of cryptocurrency trading depends on the protection of the *private key* of a cryptocurrency owner, which is the personal secret representing the ownership of and right to trade cryptocurrency, from unauthorized use by adversaries. Unfortunately, this security requirement has been violated in the common practices of owners that entrust their private keys to a remote server such as a cryptocurrency exchange for ease of use. Accordingly, our wallet application ensures that private keys are always stored and processed within MeetGo, thereby enabling secure cryptocurrency trading on a remote server without the worry of insider threats [13]. The other exemplar application we built on MeetGo is GPGPU, explained in section V. To preserve privacy and process big data efficiently, several attempts [14], [15] have been made to incorporate a GPU having massive computational power into a TEE. However, applying these attempts in practice is difficult due to their inevitable hardware modifications to the GPU itself and the associated interfaces. On the other hand, without

any further hardware changes, MeetGo can offer a trusted GPGPU computing environment by loading a GPU module implemented in the *bitstream*. Overall, MeetGo capitalizes on the programmability of FPGAs to enable users to build their own TEE on an untrusted remote server at their disposal simply by loading the corresponding hardware module in bitstreams.

In our implementation, MeetGo along with the two aforementioned applications is built on a commodity FPGA board, Xilinx VCU118, coupled with Intel x86 CPUs via the PCIe interface. According to our experiments, the core components of MeetGo are implemented with 21,393 lookup tables (LUTs), which only occupy 1.8 % of the FPGA LUT resources. It is worth noting that, MeetGo rarely affects the performance of the applications. By adopting a pipelined design, it increases the communication latency of an individual application a mere 0.72 μs. In addition, the cryptocurrency wallet is fast enough to handle 300 transactions per second, and when implementing a CNN-based image classifier, the GPGPU handles the MNIST test database in 109.11 seconds, which are both virtually zero-overhead compared to without MeetGo.

II. THREAT MODEL AND ASSUMPTIONS

Users are eager to protect the integrity and confidentiality of their code and data that are loaded on remote computing servers. For this purpose, users will employ our FPGA-based TEE to securely run their own applications and deal with security-critical data in an isolated environment. We postulate that the built-in modules of MeetGo are trustworthy, but applications dynamically installed in our TEE may contain security vulnerabilities. In this work, we consider strong adversaries (e.g., malicious insiders in charge of administration) who have full control over the host system that comes with our TEE. That is, they have no limitations in executing code, accessing data, or controlling system components, including the CPU cores and peripherals. However, note that they are completely prevented from directly reaching the inside of our TEE, which is built on top of an FPGA physically isolated from the host system. In addition, since we trust the manufacturing process of FPGAs, we assume that they have no hardware backdoors or Trojans. For this reason, the only possible attacks from adversaries are as follows. First, adversaries may attempt to undermine the isolation property of our TEE by deceiving its authentication process or by installing malicious modules inside the FPGA. They may also try to leak users' secrets by eavesdropping or interfering with communications between remote users and our TEE through varied physical and side-channel attacks. We are convinced that all the aforementioned attacks are thwarted by adopting the conventional protection mechanisms that are orthogonal to MeetGo. The details will be discussed through the security evaluation in subsection VI-C.

III. MeetGo ARCHITECTURE

In this section, we explain the architecture of MeetGo, the TEE built on an FPGA. The first step to build our TEE

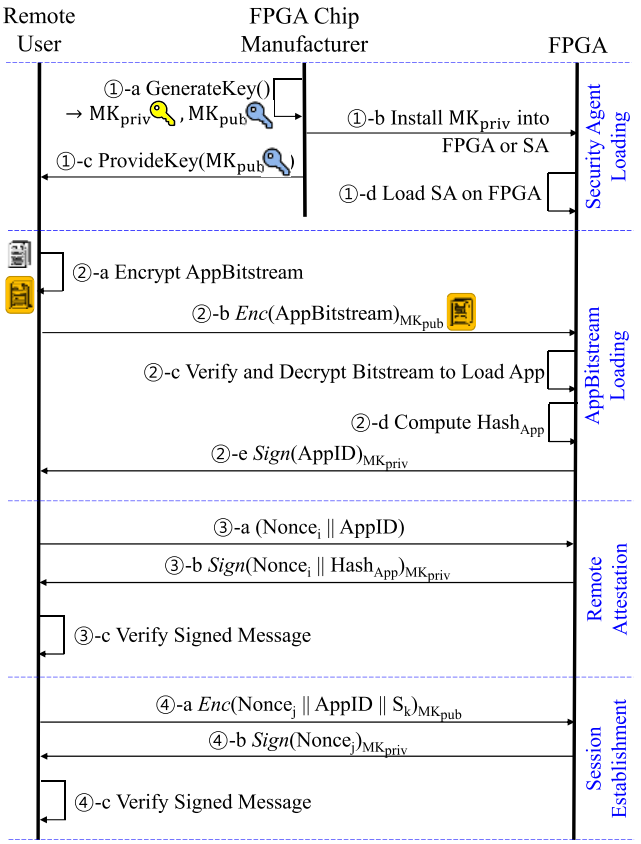

FIGURE 1. Key management for securing MeetGo.

is constructing inside the FPGA a trust anchor, which must be completely isolated from the outside. As this ultimate trust, a private key, called the *master private key* (MK_{priv}), is embedded in the FPGA. Since the chain of trust in MeetGo starts from this root of trust, we have designed our security mechanism carefully so that MK_{priv} can be tamper-proof. Along with MK_{priv}, we install a dedicated module, called the *security agent* (SA), which plays the pivotal role in our TEE. The SA, with exclusive access to MK_{priv} for various cryptographic functions, provides core security functions: (1) remote attestation that enables remote users to assess the authenticity and integrity of their applications running in the FPGA, (2) isolation to prevent unauthorized access from untrusted hosts and between applications running with private states, and (3) secure connection between a remote user and the FPGA. In the following subsections, we will explain how all these mechanisms are designed for building a TEE in the FPGA.

A. INSTALLING MASTER KEY

As depicted in ① of Figure 1, MK_{priv} is the FPGA's built-in private key from the well-known private-public asymmetric key algorithm [16], [17]. MeetGo's prerequisite is that each FPGA holds a unique private key MK_{priv}, which is essential to keep the integrity and confidentiality of our TEE built on an FPGA. It allows the FPGA to be identified through the matching public key MK_{pub}, which can be distributed to

remote users securely by certificate authorities (CAs) (①-c). We hereby posit that FPGA manufacturers who are trustworthy in our threat model in section II generate MK_{priv}/MK_{pub} pairs (①-a) and then map each pair to individual FPGAs by baking a different MK_{priv} to each FPGA during the manufacturing process. The installation of MK_{priv} (①-b) can be fulfilled by one of the following two methods, which are equally secure.

1) STORING MK_{priv} IN SA

In this method, the MK_{priv} is stored in the SA. More specifically, FPGA manufacturers implant the generated MK_{priv} inside the code of the SA (i.e., *bitstream*). As will be explained in subsection III-B, the SA bitstream is distributed and installed in encrypted form by a *secure bitstream loading mechanism* so that the MK_{priv} embedded in the SA bitstreams is securely protected against any reverse engineering attempts. In this way, however, MeetGo's essential prerequisite that each FPGA must have its own unique MK_{priv} may be unmet when identical SA bitstreams having the same MK_{priv} are loaded in multiple FPGAs. To prevent this problem, we need to strongly bind FPGAs and SA bitstreams possessing each MK_{priv}, one-to-one. This binding can be constructed by FPGA manufacturers' allocating each pair of FPGA and SA bitstream a different cryptographic key used in the secure bitstream loading mechanism.

2) STORING MK_{priv} IN NON-VOLATILE MEMORY

In this method MK_{priv} is stored in non-volatile memory such as eFUSE or PROM of FPGA rather than in SA bitstreams. However, as such internal non-volatile memory is already implemented in the current commodity FPGAs but originally intended to store the cryptographic keys used in the secure bitstream loading mechanism, FPGA manufacturers block applications from accessing the internal memory for security. Therefore, to permit the SA to reach the MK_{priv}, the FPGA hardware needs to be modified to relax this access restriction for the internal memory, but it could allow unauthorized applications to access the MK_{priv} as well. Therefore, to grant the SA exclusive access to the MK_{priv}, we can store the MK_{priv} in the internal memory in encrypted form and put the corresponding decryption key only in the SA bitstream. The SA bitstream is tamper-proof thanks to the secure bitstream loading mechanism; thereby, the decryption key and MK_{priv} are kept secure.

In a comparison of the two aforementioned methods, the former one shows a clear advantage in that it can be implemented on commodity FPGAs. On the other hand, the latter one has a drawback of requiring a slight modification on FPGA hardware. However, once the modification is done, this method has an obvious merit in terms of ease of deployment, because a single SA bitstream can be used for all FPGAs, unlike the former method that necessitates as many different SA bitstreams as the number of FPGAs. In our prototyping of MeetGo, we used the former method because we carried out the implementation on a commodity FPGA in section VI.

B. LOADING SECURITY AGENT

Based on the exclusive privilege to access MK_{priv}, the SA performs the core TEE functions such as secure application loading, remote attestation, and secure channel establishment. To prevent any interference with the loading procedure of the SA on the FPGA, the SA bitstream is designed to be loaded at boot time (see ①-d in Figure 1). It can be achieved by the *secure bitstream loading mechanism* provided on a commodity FPGA. This mechanism supports the automatic loading of a bitstream encrypted and stored inside a storage medium, such as NAND or SD card, after power-on. As briefly mentioned earlier, the decryption key is stored within non-volatile memory in the FPGA, such as eFUSE or PROM. The exclusive access to the decryption key is restricted to the built-in module of the FPGA, called the infrastructure hardware, so that even arbitrarily installed malicious applications cannot leak the key. Only the infrastructure hardware can use the key to decrypt and install the encrypted bitstream when the FPGA is booted. The SA loading process by this mechanism is logically secure, but an insider who can physically access the FPGA device may interfere with this process by switching the mode selection jumper on the FPGA board to disable this mechanism. Even so, such interference can never break the security of the SA, and it may only disturb the loading of the SA to deactivate our TEE on the FPGA. MeetGo allows remote users to check whether the TEE is activated and applications are running through a remote attestation mechanism described in subsection III-D.

C. LOADING APPLICATIONS

To load and run applications dynamically on the FPGA, we utilize the *partial reconfiguration* feature [18] that allows a subset of the FPGA to be modified by a partial bitstream downloaded while the FPGA is operating. In our design, the SA is loaded to the *static* area, which is programmable only at boot time, but applications can be loaded, whenever necessary, as partial bitstreams into the *dynamic* area that allows reprogramming at runtime. The dynamic area is again partitioned into several regions in each of which an individual application is installed. The partition is organized at boot time according to the configuration (i.e., number and size of regions) that is predefined in the SA bitstream. The partition is fixed after being configured once, and the exclusive access authority for the regions is only given to the SA, so that subsequent processes of loading applications are performed under the full control of the SA.

To be specific, the user may remotely transmit the application bitstream or request to load the bitstream stored in the server memory. In either case, to protect its contents, a bitstream is encrypted by using the asymmetric encryption algorithm using the public and private key pair, MK_{pub}/MK_{priv}, which is already associated with the FPGA, as described in subsection III-A (②-a). Note that no FPGAs have the same MK_{pub}/MK_{priv}. This implies that when the server is equipped with multiple FPGAs, a remote user may choose a specific FPGA, which the desired bitstream is targeted to run on,

simply by encrypting the bitstream with the key associated with the target FPGA. When the target FPGA receives a bitstream encrypted with MK_{pub} as input (②-b), the SA first verifies its signature to ensure that the application has been certified by a trusted third party (e.g., well-known app store) and has not been illegally modified (②-c). The SA then decrypts it with MK_{priv} and installs the bitstream on a region of its dynamic area (②-c). After successfully loading the application, the SA sends a message notifying the remote user of the application ID, together with the signature representing the integrity and authenticity of the message (②-e). As the application ID, the index of the region where the bitstream is placed within the dynamic area is used. Since, in our partial reconfiguration mechanism, the number and size of regions in the dynamic area are fixed after being configured statically at boot time and only the SA has access to the regions, an adversary cannot arbitrarily corrupt the loaded application or remap the allocated application ID to a malicious application in the operating FPGA.

The loaded application does not share any resources, such as caches and buffers, with other applications to avert the unavoidable security issues that arise under resource sharing, as stated in section I. Each application uses only the resources of its allocated region and communicates only with the SA. The SA encrypts the application response and transmits it to the remote user using a session key. It is worth noting that any entity other than the remote user sharing the session key with the SA cannot leak or tamper with the information within the encrypted message. The procedure used to securely share the session key will be described in subsection III-C.

In our design, the SA and the loaded applications are connected in accordance with the standard AXI bus protocol [19], which is one of the most popular bus protocol standards between IPs in SoC design. MeetGo hereby can incorporate various (conventional) IPs by loading them as applications, which are readily fulfilled with no interface change or a slight interface change of adding AXI bridge, which translates the original signals into AXI transactions. In this way, MeetGo can provide a GPGPU-enabled TEE, whose details are described in section V.

D. REMOTE ATTESTATION

MeetGo allows remote users to verify the authenticity and integrity of applications that are running on the FPGA at any time. The SA carries out a proof whenever there is a request for remote attestation. For this, the SA employs a static measurement scheme. Whenever the SA loads an application into a region in the dynamic area, it measures (computes the hash of) the bitstream (②-d) and stores it in a table along with the application ID (i.e., region index). Such static measurement is trustworthy, because in the design of our TEE, the SA is the only entity that is authorized to access the dynamic area where applications are installed. The attestation protocol runs as follows. The remote user can send an attestation request message for an application. The message should contain the target application's ID and an unpredictable nonce for preventing a replay attack [20] (③-a). Upon receipt, the SA responds with the measured hash value of the target application and the received nonce, after signing with the MK_{priv} (③-b). Now, the remote user can verify the attestation (③-c) by (1) checking the correctness of the nonce, (2) comparing the delivered hash value with the known value associated with the target application, and (3) authenticating the signature through the MK_{pub}.

E. ESTABLISHING SECURE COMMUNICATION CHANNEL

The SA allows the remote user to securely communicate with an application running in the FPGA. All the messages transferred between the user and the application are encrypted with a symmetric session key. To securely share this key with the FPGA, the user initiates the key exchange algorithm in the following steps: (1) The user generates a session key (S_k) and then encrypts it with the MK_{pub} of the FPGA where the target application is running. A random nonce and the application ID are also included and encrypted together (④-a). (2) When the SA receives the encrypted message, SA decrypts it with the MK_{priv} to obtain the S_k. (3) Then, the SA tries to prevent multiple users from accessing the same application at one time by managing a one-to-one mapping table between the S_k and the application ID. (4) It then returns the acknowledgement, which comprises the sent nonce and the signature generated using the MK_{priv} (④-b). (5) Finally, the remote user can be sure that a session has been opened and the associated S_k has been securely shared with the SA by verifying the signature using the MK_{pub} (④-c). It is noteworthy that unless they know the MK_{priv}, adversaries cannot launch a man-in-the-middle attack through the key exchange algorithm proposed above. We adopt AES (with Galois/Counter Mode) as our symmetric key algorithm to keep the integrity and freshness as well as the confidentiality during the communication session. When encrypted messages arrive from remote users, the SA decrypts the messages with the S_k, verifies that it is legitimate access by referring to the mapping table between the S_k and the application ID, and delivers the messages to the application running in the FPGA. In addition, the SA encrypts the response messages sent from the application with the S_k and then requests the host system to deliver the encrypted messages to the remote user. The SA performs both encryption and decryption in a pipelined manner to minimize the latency in secure communications to a little initial delays (i.e., pipeline startup delay). The S_k shared between the user and SA is valid until the user closes the established session afterward or a predefined session timeout is expired.

F. IMPLEMENTING SA

The SA consists of several internal modules each for independent operations, as illustrated in Figure 2. First, the two modules for elliptic curve cryptography (ECC) and AES are involved in the encryption and decryption processes using MK_{priv} and S_k, respectively. Next, the elliptic curve digital signature algorithm (ECDSA) module generates and verifies the signature using MK_{priv}. Then the ICAP module loads

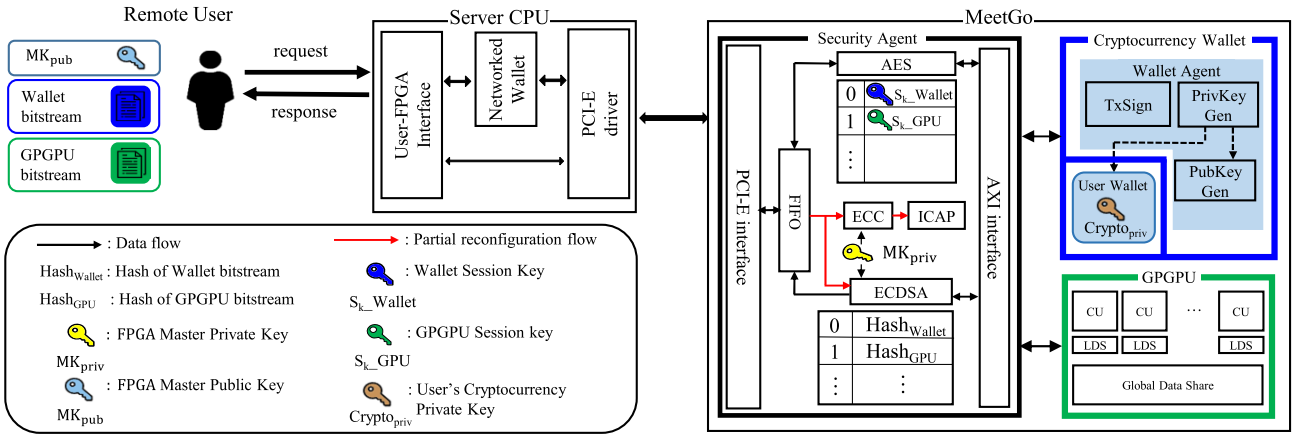

FIGURE 2. Organization of cryptocurrency wallet and GPGPU built on MeetGo.

partial bitstreams on the FPGA cooperating with the cryptographic modules. These application bistreams are verified in ECDSA, decrypted in the ECC, and configured on the FPGA by ICAP. Lastly, the two following modules are used in terms of communication: The PCIe interface module is used to communicate with the host, and the AXI interface module plays a role of connecting the SA and the applications within the FPGA.

IV. CRYPTOCURRENCY WALLET BUILT ON MeetGo

In this section, we explain our cryptocurrency wallet realized by using MeetGo. Our wallet enables users to securely carry out cryptocurrency transactions by processing their *cryptocurrency keys* (e.g., Bitcoin keys) in a TEE provided by MeetGo on a remote server. We implement our wallet to generate transactions compatible with two major currencies: Bitcoin and Ethereum.

A. WALLET IMPLEMENTATION

Figure 2 depicts the overall organization of our cryptocurrency wallet implemented on MeetGo. A cryptocurrency wallet is a program that generates and stores a pair of cryptocurrency private/public keys, denoted here as $\text{Crypto}_{\text{priv}}$ and $\text{Crypto}_{\text{pub}}$. It also executes various functions to trade cryptocurrencies. Among them, we have implemented three important functions: generating $\text{Crypto}_{\text{priv}}$, generating $\text{Crypto}_{\text{pub}}$, and signing transactions. Officially, our wallet is composed of two sub-wallets. One is the *signing-only wallet*, which performs these three functions defined in [21]. The other is the *networked wallet*, which performs the other functions, such as monitoring the spending, generating the unsigned transactions, and broadcasting the signed transactions. The signing-only wallet needs to execute functions processing $\text{Crypto}_{\text{priv}}$, the security-sensitive data that should be protected with the highest priority, whereas the networked wallet does not. Thus, in our implementation, only the former is loaded into MeetGo, and the latter is realized as software to run on the CPU. As the wallet needs to transfer data frequently to/from remote users and the signing-only wallet, we have made minor modifications to the wallet software for such communication. When a remote user wants to talk to our signing-only wallet, all the messages pass through the CPU in encrypted form via a secure channel.

In our wallet implementation, the signing-only wallet module contains the key for its user, $\text{Crypto}_{\text{priv}}$, along with three wallet functions: PrivKeyGen, PubKeyGen and TxSign. We have classified the above components as either common or user-specific components. The three functions for key and signature generation are classified as common ones, as they are performed commonly by all users. In contrast, the generated keys are classified as user-specific ones, as storage for a key must be separately assigned to each user. To efficiently utilize the FPGA resources, we generate a common module, called the *wallet agent* (WA), which performs the common signing-only wallet functions for key generation and signing. We construct a user-specific module, called the *user wallet* (UW), which contains only storage for the associated user's private key $\text{Crypto}_{\text{priv}}$. The UW associated with each remote user stores $\text{Crypto}_{\text{priv}}$ and communicates with the WA to process transaction requests. The wallet bitstreams, i.e., the WA and UW bitstreams, are verified in ECDSA, decrypted in the ECC, and configured on the FPGA by ICAP, as illustrated in Figure 2. To prevent attacks from other loaded UWs, we do not allow one UW to access another UW's interconnection with the WA. In addition, the cryptographic operations requested from UWs in the WA are performed sequentially, and all the internal buffers are flushed after handling each request.

B. OPERATING PROCEDURES OF IMPLEMENTED WALLET

After a remote user securely shares a session key with the SA, as stated in subsection III-E, the user communicates with our wallet for trading cryptocurrency using the following procedures. First, the remote user sends a request for generating $\text{Crypto}_{\text{priv}}$ and a mnemonic phrase (a seed phrase to represent the wallet) to our wallet. The mnemonic phrase should be encrypted with the shared S_k and delivered securely to the WA in the FPGA. Then the PrivKeyGen module in the

FPGA creates $\text{Crypto}_{\text{priv}}$ from the mnemonic sent from the user, according to the BIP-0039 standard [22]. The generated $\text{Crypto}_{\text{priv}}$ is stored in the UW. To keep $\text{Crypto}_{\text{priv}}$ safe, it never moves to the CPU or persistent storage. Therefore, the user needs to send the encrypted mnemonic to generate $\text{Crypto}_{\text{priv}}$ every time the procedure starts. Second, the PubKeyGen module derives the pair $\text{Crypto}_{\text{pub}}$ from $\text{Crypto}_{\text{priv}}$, and delivers $\text{Crypto}_{\text{pub}}$ to the user and the networked wallet. $\text{Crypto}_{\text{pub}}$ is used later to check the cryptocurrency account or verify the signed transaction. Third, the remote user sends a trading request to the networked wallet to generate an unsigned transaction. Then, the networked wallet checks the validity of the request (e.g., whether the balance is equal to or greater than the amount to be transferred). If the verification is successful, the TxSign module generates a signature of the transaction using the user's $\text{Crypto}_{\text{priv}}$. Finally, the SA sends the signed transaction to the networked wallet to broadcast it on the blockchain network.

V. SECURE GPGPU BUILT ON MeetGo

A noticeable advantage of MeetGo in contrast with conventional TEEs is that it can provide any hardware support desired by trusted applications for more effective execution. GPGPU is a representative type of hardware support that is considered a near necessity with the rise of machine learning and big data. However, it is not straightforward for the conventional TEEs to exploit GPGPU, since they are able to connect to this computing unit only through untrusted interfaces. For this reason, attempts have been made to harden the interfaces through hardware-level modifications [14], [15], which have a drawback in feasibility. On the other hand, the high level of programmability of the FPGA allows MeetGo to embed hardware support including GPGPU in the TEE itself; thereby, remote users can run data/computation-intensive trusted applications such as privacy-preserving neural network inferences [23], [24] in the GPGPU-enabled environment. In this section, we explain how MeetGo provides GPGPU to users and their applications.

Figure 2 depicts the overall architecture of our GPGPU implemented on MeetGo. We employed an open-core GPGPU MIAOW [25], which is available in the register-transfer level (RTL) form and prototyped in the FPGA. As explained earlier, since the SA adopts the AXI bus protocol to connect with applications, MIAOW, which is already designed for AXI, can be plugged intact into MeetGo. MIAOW is compatible with a subset of AMD's Southern Islands ISA, and it supports the OpenCL programming model widely used for general heterogeneous parallel computing. MIAOW RTL code is synthesized and implemented into the bitstream that fits into a dynamic region so that MeetGo can install it as an application upon a user's request.

Once MIAOW hardware is loaded on MeetGo, as illustrated in Figure 2, multiple compute units are instantiated as a parallel processing engine, and buffers named local data share (LDS) and global data share (GDS) are included for storing data. As mentioned earlier, since MIAOW does not need to be modified to be equipped in MeetGo, the remote user can execute the original GPU code targeted at MIAOW without tailoring it to MeetGo. The remote user transfers the code/data to the SA within the FPGA and then transmits a trigger signal to run the user's GPU code on MIAOW. Note that as explained in subsection III-E, the communication channel between the user and MIAOW is protected securely by the SA, so MeetGo can overcome the main security concern (protecting the privacy of the user's data) in a GPGPU-based service such as Machine Learning as a Service (MLaaS) [23].

TABLE 1. Synthesized results of applications on MeetGo.

FPGA modules			FPGA resources			
			LUT	FF	DSP	BRAM
MeetGo submodule	Security Agent (SA)		21,393	12,922	188	30
User Applications	Crypto-currency Wallet	PrivKeyGen	32,173	8,544	0	76
		PubKeyGen	19.225	14,047	70	76
		TxSign	75,993	32,437	110	171
		User Wallet	0	0	0	7.5
		Sum	127,391	55,028	180	330.5
	GPGPU (MIAOW)		913,521	549,836	990	1,214

VI. EVALUATION

To evaluate MeetGo, we have implemented its prototype on the Xilinx Virtex UltraScale+ VCU118 FPGA board. This development board equips the PCIe interface that transfers data from/to the host CPU at a speed of 8.0 GT/s. The 8-GB DDR4 SDRAM is available to store data in the FPGA memory.

All our FPGA modules, the two applications (cryptocurrency wallet, GPGPU), and the SA module are developed by Verilog-HDL. In our wallet application, the WA module that performs cryptographic functions for cryptocurrencies is developed from the open-source C files ([22], [26]). Those C sources are converted to HDL codes by the Xilinx Vivado HLS tool and mapped onto the FPGA as a partial bitstream. We also generated the GPGPU bitstream from the open-source MIAOW RTL, but we have slightly changed the original architecture by adding four more compute units (from 1 to 5) to improve the performance. Mainly due to the speed limit of the FPGA, the cryptocurrency wallet and SA modules are configured to operate at 25 MHz, while our GPGPU is set to be operated at 50 MHz. We have ensured that the implemented modules on the FPGA board satisfy timing constraints when a clock frequency is set to 25 MHz and 50 MHz, respectively.

A. SYNTHESIS RESULTS

Based on the parameters mentioned above, we synthesized our MeetGo design onto the FPGA, loaded our two MeetGo applications, and quantified the logics necessary for MeetGo and the applications in terms of LUTs, flip-flops (FFs), DSPs and block RAMs (BRAMs). The synthesis results are shown in Table 1. The SA occupies 1.8% (21,393/1,182,240) of the total LUTs, 0.5% (12,922/2,364,480) of the total FFs, 2.7% (188/6,840) of the total DSPs, and 1.4% (30/2,160) of

the total BRAMs. As shown from the above results, the SA module, essential in MeetGo, occupies only a small fraction of FPGA resources (e.g., 1.8% LUTs); thereby, most of the FPGA resources are still available to the applications.

The cryptocurrency wallet application module comprising the WA and UW accounts for 10.8% of the total LUTs, 2.3% of the total FFs, 2.6% of the total DSPs, and 15.3% of the total BRAMs. We deem that this size is acceptable, but we have found that if we carefully adjust the various options of the HLS tool, we can get a smaller WA module. Otherwise, experts on the RTL and cryptographic algorithm may be recruited to design those modules from scratch to attain the best area and performance. It is worth noting that these kinds of hardware updates can be applied even after it is released the the market, because FPGAs are reprogrammable.

Our GPGPU application utilizes 77.3% of the total LUTs, 23.3% of the total FFs, 45.8% of the total DSPs and 56.2% of the total BRAMs. In our current implementation, the number of compute units for parallel processing is configured to five, which is enough to provide great performance for remote users, resulting in this high ratio in FPGA utilization. When we tried to add more compute units, the timing constraint was not satisfied due to the severe routing congestion when running the place and route phase in logic synthesis. Thus, the number of compute units of GPGPU needs to be adjusted considering the FPGA chip specification and the performance requirement. Remote users should have a choice of the appropriate version of GPGPU bitstream that includes as many compute units as necessary.

B. PERFORMANCE ANALYSIS

To show the feasibility, we evaluated MeetGo in terms of performance. Experiments were performed on Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz (with 25 MB cache) with 64 GB RAM, running Ubuntu 16.04 with Linux 4.4.0.164 (64-bit). To obtain experimental results, each of the experiments was repeated at least 100 times. The power-saving mode was turned off and the CPU frequency was set to the maximum value in Linux to minimize variation between experiments.

We first measured the execution time of primitive operations of bitstream loading and data transfer between the host and MeetGo. As a result, we observed that it takes 10.31 and 61.78 minutes on average to load cryptocurrency wallet bitstream of size 43.88 MB and GPGPU bitstream of size 263.06 MB, respectively. According to our analysis, such a long loading time is attributed to the bitstream decryption process based on asymmetric key cryptography, specifically ECIES, which is one of the most robust algorithms. MeetGo by default accepts encrypted bitstreams to protect their confidentiality. This strong policy is only needed for some applications whose bitstreams statically contain secret information (e.g., cryptographic key and password). Thus, for most applications including our two examples, we can accept raw bitstreams, reducing the loading time by over 99 percent. In this case, the adversaries in the middle would be able to compromise the integrity of the bitstreams, but they will be easily detected by the remote attestation mechanism provided by MeetGo.

TABLE 2. Data transfer delay comparison.

User Applications	Data type	Data Size [Byte]	Transfer Time Delay [μs]	
			w/o MeetGo	w/ MeetGo
Cryptocurrency Wallet	Mnemonic for $Crypto_{priv}$ Generation	768	21.35	21.59
	Generated $Crypto_{pub}$	64	1.66	1.9
	Unsigned/Signed Transactions	96	2.54	2.78
GPGPU (MIAOW)	Neural-Network Parameters	56.41K	1590.57	1590.81
	User Input	3.06K	86.47	86.71
	Inference Result	4	0.22	0.46

TABLE 3. Processing delay comparison.

User Applications	Task	Processing Delay [s]		Throughput [trs/s, img/s]	
		w/o MeetGo	w/ MeetGo	w/o MeetGo	w/ MeetGo
Cryptocurrency Wallet	Signing Trans. (100k trans.)	333.41	333.63	299.93	299.73
GPGPU (MIAOW)	Classifying Img. (10k img.)	109.08	109.11	91.68	91.65

Similar to the bitstream loading, some delays are added in transferring data to applications and receiving results, because the SA in the middle protects all the data being transferred with encryption based on a symmetric key algorithm. Fortunately, the overhead is minimized thanks to the pipelining scheme, as mentioned in subsection III-E. Table 2 demonstrates that the data transfer delay of MeetGo is reasonably acceptable.

For further evaluation, we investigated the processing time of the two implemented applications when they are running on the FPGA without and with MeetGo. We developed host-side applications to give tasks and get results to the cryptographic wallet and GPGPU running on MeetGo. In particular, in regards to GPGPU, we also developed an image classification program for the MNIST database using OpenCL. This program employed the CNN model defined in [24] that we trained using Tensorflow, and executed the CNN-based inference in GPGPU built in MeetGo. Resultantly, in both applications, MeetGo incurs negligible performance overhead as clarified in Table 3. This is because MeetGo is designed not to affect the execution performance of the applications except the initial delay in secure communications. We also observed in each application that the timing variations of the PCI interface between the host and the FPGA exerts more influence on the performance than MeetGo itself.

We have also compared the performance of the two applications implemented based on MeetGo with those implemented differently in prior works: SW-only wallet [22] and GPU-based image classification [24] We note that both implementations in prior works are untrusted for the following reasons: First, privileged administrators are allowed to arbitrarily read or write the memory used by applications, as we discussed in section I, since the SW-only wallet runs within the REE. Second, the authors in [24] did not fully

TABLE 4. Performance comparison with untrusted prior work.

User Applications	Processing delay [msec]		Performance Comparison
	Untrusted prior work	MeetGo	
Signing a Cryptocurrency Transaction	0.09 [22]	3.34	37x slower
Classifying an Image	5160 [24] (0.63, batched)	10.91	473x faster (17x slower)

figure out how to secure homomorphic encryption against side-channel attacks on a commodity GPU, even though their image classification operates on encrypted data for data privacy. By comparison, MeetGo provides a more robust environment for side-channel issues (see in subsection VI-C).

As shown in Table 4, when operating a wallet function that signs a cryptocurrency transaction, our wallet running on MeetGo is on average 37 times slower than the SW-only wallet. This performance degradation is mainly attributed to the significant clock speed difference in between the CPU (2.2GHz) and the FPGA (25MHz). The reason why our MeetGo-based wallet operates with such a low clock frequency is that we produced its Verilog HDL code from C sources using the HLS tool that is still unmature to fully draw the computing power of modern FPGA boards.[1] Therefore, we believe that the operating clock will be increased if the HLS tool is improved or the wallet bitstream is developed from scratch by using Verilog HDL.

When it comes to image classification, MeetGo-based application out-performed about 473 times than the untrusted one for a single image as shown in Table 4. The authors in [24] tried to accelerate multi-image classification by enabling a batch mode where they encoded 8192 images in a form that can be processed at once. In this case, MeetGo-based classification was 17 times slower. However, since MeetGo-based one operates based on GPGPU, the batch approach would be easily applied to our implementation to enhance the performance, which is left for the future work.

C. SECURITY ANALYSIS

Although MeetGo introduces a new hardware component, the FPGA, to the system, we assert that MeetGo never widen the attack surface of the system. In the following paragraphs, we elaborate on details of potential attacks against MeetGo along with how they are tackled.

Systematically, attacks to MeetGo can be classified into those on the interface with remote users and those on MeetGo's FPGA component. First, adversaries may attempt to launch man-in-the-middle attacks to eavesdrop or corrupt the messages exchanged with remote users, but these attempts are thwarted completely because all the messages are transmitted encrypted and their freshness are easily guaranteed by adding a timestamp. Therefore, even privileged adversaries residing in the host system are prevented from disclosing or compromising the messages, and the only types of attacks they can launch are denial-of-service attacks (which are beyond the scope of our work). As for adversaries, the only way to succeed in attacks on the interface is to obtain a S_k that is used to encrypt the contents of a secure session. However, according to the key exchange algorithm in subsection III-E, the S_k is delivered from users to SA as a ciphertext that is encrypted by MK_{pub}, and MK_{priv} never leaves FPGA; thus adversaries cannot extract the S_k. Adversaries may try to pretend to be benign users and establish a secure channel with their own S_k. However, as the validity of the S_k is confined by the SA to each session, adversaries cannot exploit their S_k to meddle with other sessions (of different users/applications) already established on the interface.

Adversaries may also be able to initiate some attacks on MeetGo's FPGA component. Remember that since we assume that the SA has no security defects in our threat model, adversaries cannot manipulate this module directly. Therefore, alternatively, adversaries may try to replace the SA with a fake one that aims at performing some malicious actions such as extracting the MK_{priv} and S_k. Fortunately, this attempt is prevented thanks to the secure bitstream loading mechanism explained in subsection III-B. After power-on, this mechanism automatically loads a SA bitstream that is encrypted and authorized by FPGA manufacturers so that a fake SA cannot substitute for the genuine SA. Once it is loaded, the SA controls all access to the dynamic area where applications are loaded. Thus, adversaries are prevented from modifying already-loaded applications. Instead, they could try to load a malicious application to MeetGo, but it cannot adversely affect other applications, because all applications are isolated from each other and do not share resources such as wire, logic, or memory. Rather than tampering with bitstreams themselves, adversaries may also conduct fault injection attacks on the FPGA to maliciously alter the functioning of loaded modules (e.g., bypassing some security checks of the SA within the FPGA). They may induce physical faults by causing the power supply variations, clock glitches, electromagnetic disturbances, and so on. These fault injection attacks can be mitigated by shielding the FPGA device to protect against such physical injections. Alternatively, MeetGo can adopt the currently known mitigating techniques, including algorithmic change [27], fault detection [28], and randomization [29] techniques.

Adversaries, who are unable to directly compromise the modules in the FPGA, may try to launch side-channel attacks on MeetGo. For example, they can perform conventional side-channel attacks through shared hardware resources, such as page tables, cache units, and branch prediction-units, but MeetGo is physically isolated from these resources so that it is robust against these attacks. Besides, adversaries also can execute FPGA-specific side-channel attacks that exploit power, thermal, and co-located FPGA modules as an attack vector; however, fortunately, we can cope with them through the existing solutions [30], [31] that are orthogonal to MeetGo. Adversaries can even conduct different types of side-channel attacks by observing the timing informa-

[1] A modern FPGA board usually provides much higher clock frequency than our prototype, e.g., Xilinx VCU118 FPGA supports up to 810MHz clock frequency and Intel Stratix 10 supports 1GHz.

tion (e.g., interval and execution time) of the operations of MeetGo or the length of the messages that come from/to MeetGo. However, such timing information and message lengths that can be observed outside of the FPGA are too coarse-grained to infer meaningful information. For example, to the best of our knowledge, no attack has been found that exploits the interval, execution time, or message length only to deduce secrets from a cryptocurrency wallet. Lastly, against timing side-channel attacks, MeetGo can easily employ well-known mechanisms that eliminate timing information by regularly generating heartbeat-style packets [32].

VII. DISCUSSION

A. SUPPORT FOR VARIOUS ARCHITECTURES

In addition to GPGPU, MeetGo can fully utilize the programmability of the FPGA to provide remote users with a TEE based on different types of architecture. For example, open-source CPUs like RISC-V [33] and ARM Cortex-M series [34] can be readily converted to bitstreams that are loadable by MeetGo. Therefore, by installing these bitstreams onto MeetGo, remote users can execute trusted applications compiled with the ISA of RISC-V or ARM Cortex-M. It would be particularly beneficial, for example, for a remote user with Cortex-M based IoT devices wanting to migrate their sensitive code to MeetGo to run it securely in the server side containing the necessary data.

B. EASE OF APPLICATION DEVELOPMENT

With the advance of the high-level synthesis (HLS) technique, it is not necessary to use a low-level language like Verilog-HDL to develop applications of MeetGo. HLS allows developers to work at a higher abstraction level. HLS has been studied in-depth for the past decade, and the commercial HLS tool has also recently achieved convincing levels in terms of area, power, and performance [35]. Moreover, major FPGA vendors, Xilinx [36] and Intel [37] are officially supporting OpenCL, the standard heterogeneous programming language, making it easier for developers to exploit the highly parallel nature of FPGAs. To sum up, the entry barriers for the development of FPGA applications have been lowering.

C. FEASIBILITY ON MOBILE DEVICES

Since MeetGo is designed to operate independently to CPU architecture or OS, it can provide the same TEE environment to the mobile platform in the same way as the server system. The only requirement to apply MeetGo to mobile platforms is that the FPGA should be mounted. For this purpose, a one-chip or two-chip solution that packs the CPU and FPGA in the same die or that mounts the CPU and FPGA on separate dies is possible. Here, the spec of the FPGA chip to be incorporated into the mobile platform can be selected flexibly according to the targeted power/area constraint.

VIII. RELATED WORK

To protect trusted applications from the REE under the control of system administrators, various hardware-based TEEs have been developed by extending existing CPU architectures. Among them, probably the closest TEE model to MeetGo would be Intel *SGX* [38], which has been integrated into commodity desktop CPUs since Intel Skylake in 2015. Like MeetGo, SGX can be used to create and support a secure runtime environment, called the *enclave*, for the remote application of any user who wants to run a program remotely on the server equipped with SGX-enabled CPUs. Each enclave is isolated by the underlying hardware from any entities in the server, including the OS and hypervisor, such that all private user data inside the enclave can be protected against any theft and tampering attempts. Also like MeetGo, SGX enables remote users to attest their applications inside enclaves by offering attestation key infrastructures. In academia, there have been similar efforts to incorporate SGX-like TEEs into the RISC-V open-source CPU [39], [40].

One notable difference between MeetGo and all these CPU extensions for TEE is that their implementations are only available in certain CPU platforms while ours is applicable to any platform based on the CPU/FPGA hybrid architecture [3]–[5]. Another difference is that as their TEEs are built in the host CPU hardware, applications running inside the TEEs share many computing resources with the untrusted host system. For instance, an SGX enclave uses page tables, caches, and branch prediction units that are all accessible or shared by untrusted entities like the OS or other enclaves. The potential problem of such resource sharing is that it increases the chances of user secrets inside the enclave being leaked by side-channel attacks [41], [42]. In the development of MeetGo, therefore, we endeavored to avoid the resource sharing problem by capitalizing on the reconfigurability of the FPGA in that we implemented into MeetGo all hardware modules necessary to perform sensitive cryptocurrency transactions and execute security-critical GPGPU programs, as described in section IV and section V. Furthermore, to avert attacks from other applications, in the implementation, we ensured that no resources were shared among MeetGo applications.

Aside from this one, several studies have been performed noting the FPGA's value as a TEE isolated at the hardware level. However, most of them [43]–[45] have focused on building a TEE for local users, unlike MeetGo, which is specialized for remote users. For example, a state-of-the-art work [45] proposed a FPGA-based TEE whose trustworthiness is ensured on the basis of its self-provisioned master key. However, this work is less appropriate than MeetGo for remote users at far distances because it requires the the users to sign their applications through a reliable channel (e.g., physical access to the FPGA) prior to installing them to the FPGA.

IX. CONCLUSION

MeetGo is an ordinary FPGA on a modern computer armed with a security mechanism for remote computing. Our security mechanism employs cryptographic algorithms based on the master key prudently managed in the FPGA.

These algorithms are used not only to construct a TEE for remote users, but also to establish a secure communication channel between the remote users and their applications running inside the TEE. In the actual implementation, MeetGo has been used to (1) create cryptocurrency wallets that enable the owners to trade their currencies remotely on a server without any server-side intervention and (2) provide GPGPU, which securely accelerates data-intensive computations without reliance on the legacy GPU. Experimentally, MeetGo showed a low FPGA resource utilization ratio and incurred negligible performance overhead on those applications. We also analyzed potential attacks targeting MeetGo and explained why MeetGo is safe from these attacks.

REFERENCES

[1] R. F. Trzeciak. (2017). *Sei Cyber Minute: Insider Threats*. Accessed: May 26, 2020. [Online]. Available: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=496626

[2] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, Aug. 2015, pp. 57–64.

[3] Intel. (2018). *Intel Xeon Gold 6138 Processor*. Accessed: May 26, 2020. [Online]. Available: https://en.wikichip.org/wiki/intel/xeon_gold/6138p

[4] Amazon. (2018). *Aws Ec2 FPGA Development Kit*. Accessed: May 26, 2020. [Online]. Available: https://github.com/aws/aws-fpga

[5] Intel. (2018). *Intel Programmable Acceleration Card With Intel Arria 10 Gx FPGA Datasheet*. Accessed: May 26, 2020. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literatur%e/ds/ds-pac-a10.pdf

[6] H. Oh, A. Ahmad, S. Park, B. Lee, and Y. Paek, "TRUSTORE: Side-channel resistant storage for SGX using intel hybrid CPU-FPGA," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2020, pp. 1903–1918, doi: 10.1145/3372297.3417265.

[7] K. Neshatpour, M. Malik, M. A. Ghodrat, and H. Homayoun, "Accelerating big data analytics using FPGAs," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2015, p. 164.

[8] R. Dhanabal, S. K. Sahoo, V. Bharathi, K. Dowluri, B. S. R. P. Varma, and V. Sasiraju, "FPGA based image processing unit usage in coin detection and counting," in *Proc. Int. Conf. Circuits, Power Comput. Technol. [ICCPCT]*, Mar. 2015, pp. 1–5.

[9] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A survey of FPGA-based neural network accelerator," 2017, *arXiv:1712.08934*. [Online]. Available: http://arxiv.org/abs/1712.08934

[10] B. Nagy, P. Orosz, and P. Varga, "Low-reaction time FPGA-based DDoS detector," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp. (NOMS)*, Apr. 2018, pp. 1–2.

[11] A. Boutros, B. Grady, M. Abbas, and P. Chow, "Build fast, trade fast: FPGA-based high-frequency trading using high-level synthesis," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Dec. 2017, pp. 1–6.

[12] A. Surendar, "Fpga based parallel computation techniques for bioinformatics applications," *Int. J. Res. Pharmaceutical Sci.*, vol. 8, pp. 124–128, 01 2017.

[13] T. Dvorin. (2019). *Crypto Hacks: The Rise of the Rogue Insider*. Accessed: May 26, 2020. [Online]. Available: https://www.unboundtech.com/crypto-hacks-the-rise-of-the-rogue-insider

[14] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on GPUs," in *Proc. 12th USENIX Conf. Operating Syst. Design Implement. (OSDI)*, Berkeley, CA, USA: USENIX Association, 2018, pp. 681–696.

[15] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity GPUs," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 455–468, doi: 10.1145/3297858.3304021.

[16] *Digital Signature Standard (DSS)*, National Institute of Standards and Technology, FIPS Publication, Gaithersburg, MD, USA, May 1994.

[17] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[18] Xilinx. (2020). *Vivado Design Suite User Guide*. Accessed: May 26, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug%901-vivado-synthesis.pdf

[19] ARM. (2011). *Amba(R) Axi and Ace Protocol Specification*. Accessed: Aug. 22, 2019. [Online]. Available: http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI%4_specification.pdf

[20] P. Syverson, "A taxonomy of replay attacks [cryptographic protocols]," in *Proc. Comput. Secur. Found. Workshop VII*, Jun. 1994, pp. 187–191.

[21] B. Project. (2019). *Wallets*. Accessed: May 28, 2020. [Online]. Available: https://developer.bitcoin.org/devguide/wallets

[22] Libbitcoin. (2019). *Bip-0039 Standard*. Accessed: May 28, 2020. [Online]. Available: https://github.com/libbitcoin/libbitcoin-system/blob/master/src/wallet

[23] E. Hesamifard, H. Takabi, M. Ghasemi, and R. N. Wright, "Privacy-preserving machine learning as a service," *Proc. Privacy Enhancing Technol.*, vol. 2018, no. 3, pp. 123–142, Jun. 2018. [Online]. Available: https://content.sciendo.com/view/journals/popets/2018/3/article-p123.xm%l

[24] A. Al Badawi, J. Chao, J. Lin, C. F. Mun, J. J. Sim, B. H. M. Tan, X. Nan, K. M. M. Aung, and V. Ramaseshan Chandrasekhar, "Towards the AlexNet moment for homomorphic encryption: HCNN, theFirst homomorphic CNN on encrypted data with GPUs," 2018, *arXiv:1811.00778*. [Online]. Available: http://arxiv.org/abs/1811.00778

[25] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, and K. Sankaralingam, "Enabling GPGPU low-level hardware explorations with MIAOW: An open-source RTL implementation of a GPGPU," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, pp. 1–21, Jul. 2015, doi: 10.1145/2764908.

[26] B. Core. (2019). *secp256k1*. Accessed: May 28, 2020. [Online]. Available: https://github.com/bitcoin/bitcoin/tree/master/src/secp256k1

[27] R. Gennaro, A. Lysyanskaya, T. Malkin, S. Micali, and T. Rabin, "Algorithmic tamper-proof (ATP) security: Theoretical foundations for security against hardware tampering," in *Theory Cryptography*, M. Naor, Ed. Berlin, Germany: Springer, 2004, pp. 258–277.

[28] H. Mestiri, N. Benhadjyoussef, M. Machhout, and R. Tourki, "An FPGA implementation of the AES with fault detection countermeasure," in *Proc. Int. Conf. Control, Decis. Inf. Technol. (CoDIT)*, May 2013, pp. 264–270.

[29] A. Shamir, "Method and apparatus for protecting public key schemes from timing and fault attacks," U.S. Patent 5 991 415 A, Nov. 23, 1999.

[30] J. Wu, Y. Shi, and M. Choi, "FPGA-based measurement and evaluation of power analysis attack resistant asynchronous S-box," in *Proc. IEEE Int. Instrum. Meas. Technol. Conf.*, May 2011, pp. 1–6.

[31] J. Knechtel and O. Sinanoglu, "On mitigation of side-channel attacks in 3D ICs: Decorrelating thermal patterns from power and activity," in *Proc. 54th Annu. Design Autom. Conf.*, Jun. 2017, pp. 1–6.

[32] S. Aga and S. Narayanasamy, "InvisiMem: Smart memory defenses for memory bus side channel," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 94–106.

[33] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, and S. Karandikar, "The rocket chip generator," EECS Dept. Univ. California Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-17, 2016.

[34] ARM. (2020). *Easy access to cortex-m processors on FPGA*. Accessed: May 26, 2020. [Online]. Available: https://www.arm.com/resources/designstart/designstart-fpga

[35] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.

[36] Xilinx. (2019). *Sdaccel Environment Userguide*. Accessed: May 26, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug%1023-sdaccel-user-guide.pdf

[37] Intel. (2019). *Intel(R) FPGA Sdk for Opencl(TM) Pro Edition*. Accessed: May 26, 2019. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literatur%e/hb/opencl-sdk/aocl_getting_started.pdf

[38] V. Costan and S. Devadas, "Intel SGX explained," IACR Cryptol. ePrint Arch., Tech. Rep. 2016/086, 2016.

[39] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *Proc. 25th USENIX Secur. Symp. (USENIX Secur.)*, 2016, pp. 857–874.

[40] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanovic, "Keystone: A framework for architecting tees," 2019, *arXiv:1907.10119*. [Online]. Available: https://arxiv.org/abs/1907.10119
[41] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *Proc. 11th USENIX Workshop Offensive Technol. (WOOT)*, 2017, p. 11.
[42] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.*, 2017, pp. 557–574.
[43] O. Gelbart, P. Ott, B. Narahari, R. Simha, A. Choudhary, and J. Zambreno, "Codesseal: Compiler/FPGA approach to secure applications," in *Intelligence and Security Informatics*, P. Kantor, G. Muresan, F. Roberts, D. D. Zeng, F.-Y. Wang, H. Chen, and R. C. Merkle, Eds. Berlin, Germany: Springer, 2005, pp. 530–535.
[44] E. M. Benhani, L. Bossuet, and A. Aubert, "The security of ARM TrustZone in a FPGA-based SoC," *IEEE Trans. Comput.*, vol. 68, no. 8, pp. 1238–1248, Aug. 2019.
[45] A. Coughlin, G. Cusack, J. Wampler, E. Keller, and E. Wustrow, "Breaking the trust dependence on third party processes for reconfigurable secure hardware," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2019, pp. 282–291, doi: 10.1145/3289602.3293895.

HYUNYOUNG OH received the B.S. and M.S. degrees in electrical and electronic engineering from Yonsei University, South Korea, in 2005 and 2007, respectively. He is currently pursuing the Ph.D. degree in electrical and computing engineering with Seoul National University, South Korea. He worked as a SoC Designer with Samsung Electronics Company Ltd., South Korea, from 2007 to 2017. His research interest includes hardware-backed system security against various types of threats.

KEVIN NAM received the B.S. degree in electrical and computer engineering from Seoul National University, South Korea, in 2020, where he is currently pursuing the Ph.D. degree in electrical and computing engineering. His research interest includes hardware-backed system security against various types of threats.

SEONGIL JEON received the B.S. degree from the School of Electronic Engineering, Soongsil University, South Korea, in 2017, and the M.S. degree in electrical and computing engineering from Seoul National University, South Korea, in 2020. His research interest includes hardware-backed system security against various types of threats.

YEONGPIL CHO received the B.S. degree in electrical engineering from POSTECH, South Korea, in 2010, and the Ph.D. degree in electrical and computer engineering from Seoul National University, South Korea, in 2018. He is currently a Professor with the Department of Computer Science, Hanyang University. His research interest includes system security against various types of threats.

YUNHEUNG PAEK (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from Seoul National University, South Korea, in 1988 and 1990, respectively, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, in 1997. He is currently a Professor with the Department of Electrical and Computer Engineering, Seoul National University. His research interests include system security with hardware, secure processor design against various types of threats, and machine learning based security solution.