FPGAS FOR TRUSTED CLOUD COMPUTING

Ken Eguro

Embedded and Reconfigurable Computing
Microsoft Research
Redmond, WA USA
email: eguro@microsoft.com

Ramarathnam Venkatesan

Cryptography Security and Applied Mathematics
Microsoft Research
Bangalore India and Redmond, WA USA
email: venkie@microsoft.com

ABSTRACT

FPGA manufacturers have offered devices with bitstream protection for a number of years. This feature is currently primarily used to prevent IP piracy through cloning. However, in this paper we describe how protected bitstreams can also be used to create a root of trust for the clients of cloud computing services. Unlike related software-based solutions, this hardware-based approach solves a fundamental problem that currently impedes the greater adoption of cloud computing: how to secure client data and computation from both potential external attackers and an untrusted system administrator. We examine how this approach can be applied to the specific application of handling sensitive health data. This system maintains the advantages of the cloud with minimal additional hardware. We also describe how this system can be extended to provide a more generic secure cloud architecture.

1. INTRODUCTION

Cloud computing services offer many advantages for potential customers: low startup cost, high-availability, instant access to massive computing power, no need for in-house technical expertise, etc. That said, applications that deal with sensitive data present a significant problem for the existing cloud computing paradigm in which client applications run within a virtual machine on public cloud servers.

From the client's viewpoint, they may be hesitant to place this type of data on a publically accessibly system to which they do not have exclusive and ultimate administrator control. For example, although encryption can protect the data while in transit to the datacenter or while at rest in cloud storage, accessing sensitive data while it is actively being used in existing software-based cloud machines is as simple as attaching a debugger (or equivalent) to the process, virtual machine, or hypervisor. While this type of behavior would not be intentionally performed by the service provider, it is theoretically possible. More importantly, the "access-from-anywhere" and high-scale load balancing philosophies of the cloud naturally subjects the machines to a multitude of potential problems such as viruses and other malware. Thus, clients with sensitive data need more explicit guarantees regarding the security of their computations and data.

Stronger guarantees are also advantageous from the cloud operator's standpoint because it limits their liability. For example, even if a leak occurs on a machine outside of the cloud, blame may be placed on some latent vulnerability of the cloud machines or even upon a member of the cloud administration staff – with little recourse to prove otherwise.

In this paper we describe a system that addresses this problem using FPGAs. As we will discuss, due to their fundamental characteristics, FPGAs offer a substantially smaller and more well-defined attack surface as compared to traditional software-based systems. This allows us to make stronger security guarantees under more robust attack models.

The FPGAs are programmed to form a flexible, independent trusted third party compute platform within the cloud infrastructure. Since these devices run as autonomous compute elements, the cloud administrator does not have low-level access to computations running within the FPGA. Clients can offload sensitive parts of their applications to these devices. This in-cloud hardware offloading avoids potential vulnerabilities in the software stack and eliminates the performance and other issues associated with hosting sensitive parts of applications in client machines outside of the cloud.

2. CASE STUDY: MEDICAL RECORDS

To illustrate the issues that face potential cloud computing applications, consider a system for storing and processing medical data. Shown in Fig. 1, patient information is stored in a database and mined for statistical information (e.g. for a pharmaceutical drug trial). The computational complexity of mining large databases would make this problem well-suited to existing cloud solutions.

However, the personal nature of the patient information that needs to be stored creates security concerns that are not addressed in today's cloud systems. These issues stem from the fact that the database must be able to link health and personally identifiable information (PII) for each patient (e.g. the patient name and treatment). There are strict laws governing applications that store this type of information together (e.g. the Health Insurance Portability and Accountability Act, or HIPAA).

A common way to protect this information is through *tokenization* and *encryption*. Shown in Fig. 1a, as soon as patient data enters the system, all PII is shunted to a process that anonymizes it into unique tokens that are stored in the database. This PII is also encrypted to allow later retrieval, but most computations, such as those needed for data mining, can be performed using just the tokens. Only a small fraction of processing, such as bill generation, typically needs the ciphertext or plaintext that the tokens represent.

Note that compliance regulations such as HIPAA only require PII to be held securely. The rest of the data can be held as plaintext. For example, it is acceptable to store the fact that a patient has a particular condition as plaintext, as long as the patient's name is kept tokenized or encrypted.

Also note that the tokenized and encrypted data is only as secure as our handling of the plaintext data before it is tokenized/encrypted and the security of the tokenization/ encryption process itself. For this reason, when patient data is initially uploaded from patients or doctors, it will arrive at the servers via a secure protocol (e.g. SSL). However, this encryption only protects the data while it is in transit. After the data arrives on the server, it is delivered as plaintext to the tokenization and encryption process. This plaintext input, along with the keys necessary for encryption will be resident in the machine when the data is tokenized and encrypted.

As shown in Fig. 1b, the security concerns regarding performing this operation in cloud machines may cause clients to continue hosting part of their application in traditional, privately-held servers. In this example, although the client can leverage the computational and storage power of the cloud for the database and analytics, they must still maintain one or more local servers to perform tokenization and encryption. Unfortunately, in this case many of the advantages of the cloud are nullified. Clients still need to have local technical know-how and infrastructure. Furthermore, applications can suffer severe performance issues because inter-site communication is much slower than intra-site communication.

3. THE NEED FOR HARDWARE-BASED TRUSTED COMPUTING MODULES

As shown in Fig. 1c, the security concerns surrounding the visibility of sensitive data and the integrity of sensitive computations to attackers can be alleviated by offering trusted compute resources within the cloud. These discrete trusted computing nodes will offer strong security guarantees unavailable in normal cloud servers.

Segregating sensitive computations not only makes sense because it naturally reduces the likelihood for outside interaction and interference, these nodes may need to make design compromises in the name of security that would not be appropriate given the high-scale requirements of general-purpose cloud machines. For example, a server running in single-user mode is fundamentally more secure than one running in multi-user mode, but this drastically reduces the utility of the machine.

The goal here is for the cloud operator to be able to provide security as part of a Service Level Agreement (SLA). In today's cloud systems, customers can be granted SLAs with guaranteed minimum characteristics such as network bandwidth or CPU time. These SLAs are possible because the system can throttle other users and give specific customers a provable amount of resources, within a degree of mathematical certainty, assuming an appropriate model to account for hardware failure.

For service providers to offer SLAs for security, the trusted compute resources must be able to offer similar strong guarantees for security, assuming an appropriate attack model. This can be accomplished if the trusted computing device has certain capabilities:

a) Store a key.
b) Decrypt, authenticate and load binaries sent to the device.
c) Decrypt and authenticate data sent to the device.
d) Perform the computations exactly as prescribed by authenticated binaries on authenticated data.

As will be discussed in more detail in Section 5, all other operations that trusted compute nodes need to perform can be derived from these four base capabilities.

These capabilities are subject to the assumptions made by the attack model. Our attack model assumes that the following operations are sufficiently difficult that they are effectively impossible in practice:

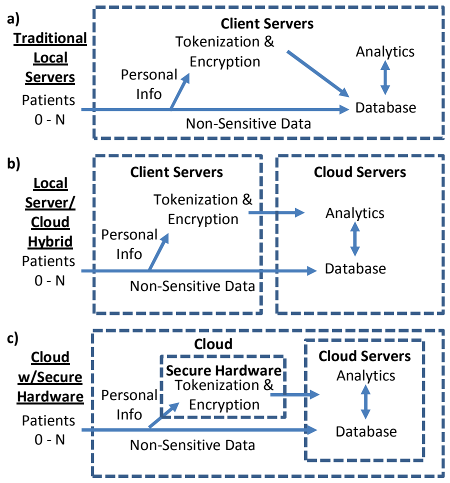


Fig. 1. Architectures for storing sensitive data

a) Breaking the cryptography used. Without access to keys, we cannot decrypt encrypted and signed binaries or data. Also, we cannot create or alter existing encrypted and signed binaries or data.
b) Loading a binary that cannot be decrypted and authenticated properly.
c) Retrieving binary or state information on the device from outside the device. Unless a currently running computation explicitly sends data out of the device, it will remain unknown.
d) Altering the behavior of a loaded binary.
e) Altering data currently on the device.

It is currently difficult for conventional software-based cloud servers to offer sufficiently strong guarantees under this security model. This is not to say that it is impossible build a software-based system to meet an acceptable security bar under this attack model. However, it is likely that a hardware-based system could be designed in a more elegant and easily verified manner, providing less opportunity for attackers with superior performance.

For example, conventional processors rely on a single, physically unified memory space for both program and data. Thus, vulnerabilities such as buffer overrun or rootkits that can defeat memory protection can alter program memory at runtime. This violates assumptions *d* and *e* in this attack model.

In contrast, FPGAs can offer highly isolated memory spaces. For example, if a computation is directly implemented in logic, the "program" space defining that computation is in the configuration of the LUTs, FFs, and routing fabric. On the other hand, the "data" space is represented in the contents of the block memory and FFs. Aside from well-defined structures like the Xilinx ICAP (and possibly unusual configurations such as using a LUT in both memory and logic modes), there is no way for values to migrate between one memory space and the other.

In a similar vein, from a performance standpoint it is highly desirable that a system concurrently handle multiple tasks. This might be as basic as the ability to overlap I/O and compute. Because processors fundamentally execute sequentially, operating systems need to be relatively complex to offer this type of feature. The operating system must be able to timeslice multiple live processes that all share a single physical memory.

Customized hardware on the other hand is naturally parallel. This functionality can be easily implemented with independent circuits, each with their own state machines and memories.

4. RELATED WORK

There has been a large volume of previous work in related areas, attempting to combat the limitations of traditional cloud servers. However, all of these approaches either have security or practical limitations that make them unattractive for use in the cloud.

For example, the Trusted Platform Module (TPM) [2] offers a small suite of functionality on conventional processor-based machines to provide features such as authenticated boot. However, this system assumes a much weaker attack model than we use here. For example, although software binaries are authenticated when loaded, this is not sufficient to defend against modifications that might be made at runtime, such as viruses. This violates assumptions *d* and *e* in our attack model.

Similarly, the TPM does not offer the capability to perform encryption or decryption locally within the device. Instead, it transfers keys stored within the device into the machine's main memory. At this point, it relies on the processor to use this key to perform encryption or decryption. Again, this relies on the BIOS or software to protect the regions of memory that contain the key. This protection cannot withstand attacks such as cold boot, violating assumption *c* in our attack model.

The work in [4] implements a full system, including a processor and a TPM inside an FPGA. This full integration solves many of the problems of more traditional TPM-based systems, including reducing the feasibility of cold-boot attacks. This work is the most similar to the concepts in this paper, but our focus is different. [4] focuses on bringing the full suite of TPM functionality to a soft processor running on the FPGA. This paper streamlines the system, offering only the bare functionality necessary to uniquely identify and communicate with a computation implemented directly in the logic fabric.

Another approach that is used is to segregate sensitive applications into special-purpose high-security servers. These machines must be isolated, both logically (in terms of an independently firewalled network) and physically (using sequestered racks with security cages and cameras). Rather than a more general-purpose operating system, these high-security servers run small-footprint purpose-built software stacks. However, this approach profoundly breaks the business model of the cloud. For example, such machines cannot scale due to the necessary physical isolation constraints, they are not generically swappable for failover or load balancing, and they may require external third-party administration.

Secure co-processor (SCP) expansion cards such as those available from IBM [7] or smartcards [6] have also been used to provide secure computing in untrusted servers. These completely self-contained devices can be installed inside existing machines, eliminate some of the problems associated with high-security servers, such as maintaining separate secure facilities. Unfortunately, existing SCPs and smartcards are built for highly specific low performance applications such as use in cash machines. This makes them inappropriate for use in high-scale cloud systems.

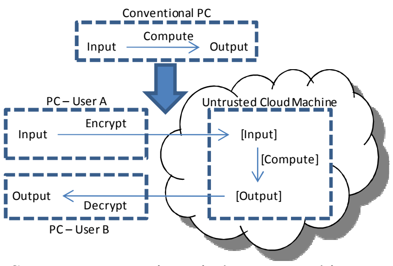


Fig. 2. Secure computation via homomorphic encryption

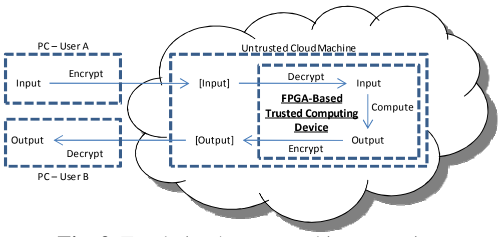


Fig. 3. Emulating homomorphic encryption

Hardware security modules (HSMs), such as those produced by Thales [11], are monolithic security expansion cards like SCPs and smartcards, but are built from dedicated logic rather than low-frequency embedded processors. This allows them to generally offer better performance. At the same time, though, they are also sold as black-box purpose-built appliances. Thus, different applications will generally require different cards. Again, this creates migration, scaling and failover problems. Furthermore, since server farms generally operate lights-out, installing new cards to support new applications presents a serious logistical problem.

As we will discuss in Section 5, one way of looking at the approach proposed in this paper is that FPGAs can be used to build "virtual" HSMs. Essentially, we can install a single programmable hardware device and dynamically swap out encrypted and signed bitstreams to support an unlimited number of existing and future applications without any of the management and logistical problems of existing HSMs.

In a completely different vein, homomorphic encryption [1, 5] promises to circumvent these issues by pushing the entire burden of security into the cryptography that is used. For example, as seen in Fig. 2, a fully homomorphic cryptographic system would allow arbitrary operations to be performed directly on encrypted data. In this example, there is a desired operation, *Compute*, that is normally performed on plaintext input and produces plaintext output (top of Fig. 2). This can be replaced by a homomorphic function, [*Compute*], that can be executed in an untrusted computation platform (bottom of Fig. 2). The homomorphic function does not require any keys to perform the operation. In this way, the risk to security can be minimized.

That said, to date, no computationally tractable homomorphic encryption algorithm has been developed [1]. Since scalable performance is a key feature of the cloud, we cannot use any cryptographic system that creates serious performance issues. Furthermore, most existing homomorphic encryption algorithms are based on relatively new cryptographic operations (e.g. bilinear pairing or *ad hoc* polynomial approaches). These have not been as well vetted by the cryptography community as compared to the operations used in more conventional algorithms such as RSA, SHA, and AES.

As we will discuss in Section 5, the system we propose in this paper effectively emulates the behavior of homomorphic encryption by providing a protected area within the untrusted environment in which sensitive operations can be performed securely. As seen in Fig. 3, if users were confident that their private data could be safely transmitted into this cordoned-off region and manipulated there without being observed by any other part of the system, including the system administrator, the trusted computing device could simply decrypt the ciphertext, perform the conventional *Compute* operation, and re-encrypt the results.

5. SYSTEM ARCHITECTURE

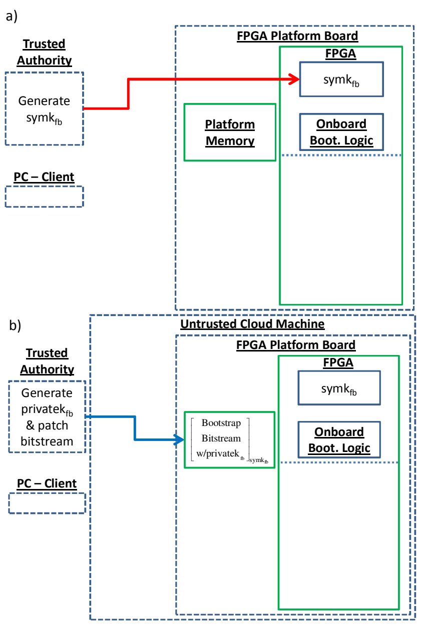
There are three distinct phases for using our FPGA-based trusted compute platform: key/infrastructure setup, user application setup/operation, and system updates.

5.1. Key and Infrastructure Setup

Deployment of the trusted computing nodes begins with a trusted authority (TA). All potential clients and the cloud operator trust the TA. Before the FPGA platform is delivered to the cloud operator, as seen in Fig. 4a, the TA generates a random symmetric encryption key, $symk_{fb}$. This symmetric key is copied into the onboard key memory of the FPGA.

As discussed in [3], modern FPGAs such as the Xilinx Virtex-6 contain onboard key memory and bootstrapping logic. The key memory can only be written from an external port on the FPGA. Similarly, the contents can only be read through a dedicated connection to onboard bootstrapping logic. Thus, after the trusted authority places a key into the onboard key memory on the device, it cannot be read externally. The key memory may be externally over-writable (e.g. battery-backed RAM or flash), but not externally readable.

Since this key is the most fundamental link in the system and will protect all future interactions, copying the key to the FPGA must be done in a secure location, likely on the TA's premises. The necessity for a private transfer is denoted in Fig. 4a with a red arrow. After the key has



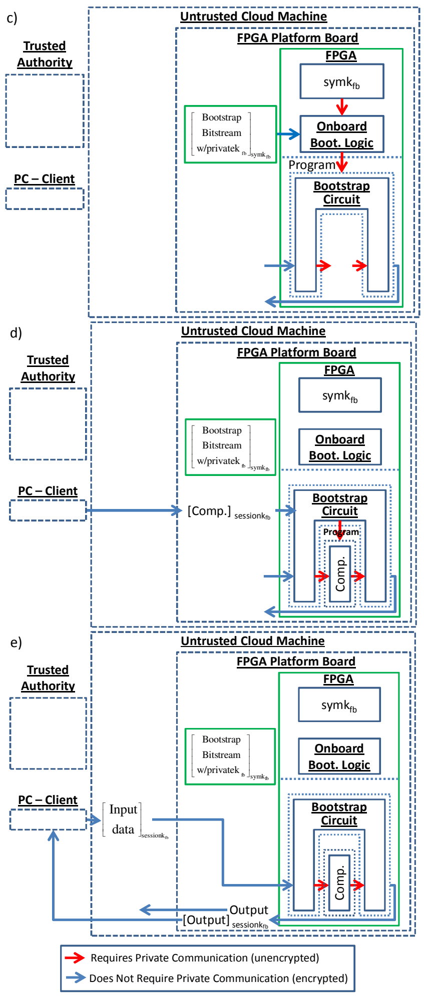


Fig. 4. System architecture, setup, operation and update

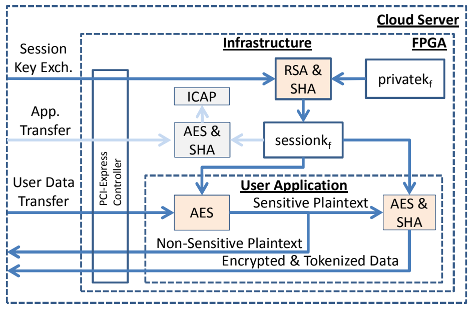


Fig. 5. FPGA system infrastructure and user application

been written, the FPGA can be delivered to the cloud operator and installed. In our system, the FPGA is installed into an 8x PCI-Express slot in a commodity cloud server.

At this point, the FPGA can be used as a virtual HSM. For example, if the TA is highly accessible and clients are willing to send their applications to the TA, the TA can simply encrypt and sign binaries with the $symk_{fb}$ of each individual FPGA card. After this, these applications can be decrypted and loaded on demand onto the reconfigurable platform. This very simple approach is also attractive if the set of necessary applications is relatively static (i.e. a fixed suite of secure cloud services).

Loading encrypted binaries is secure because, as shown in Fig. 4, modern FPGAs contain dedicated onboard bootstrapping logic built directly into the device by the manufacturer. Unlike the majority of the compute resources on the FPGA, this circuitry is not customizable. Its sole purpose is to read an encrypted configuration bitstream from the external platform memory module and decrypt/verify it with AES and HMAC using $symk_{fb}$ stored in the onboard key memory. If the key in the onboard key memory and encrypted bitstream "match" (i.e. the bitstream was encrypted with the key), the resulting bitstream will be valid and it will be used to program the configurable region of the FPGA. If the key and encrypted bitstream do not "match" (i.e. the bitstream was encrypted with a different key), the resulting bitstream will not be valid and the FPGA will raise an error signal indicating a problem with configuration.

Although the basic model of virtual HSMs is useful by itself, we would also like to support a more sophisticated operational model does not require direct TA involvement for each and every bitstream. Towards that end, the TA can produce a single generic bootstrapping binary for each FPGA card that acts as onboard infrastructure that can receive and load client applications directly.

To accomplish this, as shown in Fig. 4b, the TA generates a public/private RSA key pair unique to each device, places the private key (*privatek$_f$*) into the bootstrapping binary, and publishes the public key via standard public key infrastructure (PKI). As shown in Fig. 5, in our proof-of concept system, this bootstrapping binary contains a PCI Express controller used to link the FPGA with the host server, an RSA/SHA core to negotiate a symmetric session key exchange with clients, and an AES/SHA core to decrypt and authenticate communicate with clients.

The TA then encrypts the bootstrapping bitstream with AES using *symk$_{fb}$* and transfers it to the platform flash on the FPGA board. As shown in in Fig. 4b, since the bitstream is already encrypted, this transfer does not need to be protected further. This is denoted with a blue arrow. The TA can send this bootstrapping bitstream to the cloud operator, who programs it into the platform memory. As shown in Fig. 4c, when power is applied to the FPGA after installation, the device enters its normal boot sequence.

5.2. User Application Setup and Operation

After the bootstrap bitstream is running on an FPGA in the cloud, the client (or a developer that the client trusts) can create an application for the FPGA to handle sensitive data.

When the client would like to perform a sensitive operation, they request an FPGA from the cloud service. The cloud service provisions an FPGA and the client can connect to this device to load their application securely using standard PKI, similar to an SSH session. The client will use the public key of the device to exchange a symmetric session key, *sessionk$_f$*.

In our case study, the client application performs tokenization/encryption. As shown in Fig. 5, an AES core in the client application decrypts incoming data with *sessionk$_f$*. Non-sensitive data is passed back to the software-based server and sensitive data is passed to an AES/SHA core. This core uses *sessionk$_f$* to produce encrypted and tokenized data.

In a system with full flexibility, the client would be able to compile their FPGA code locally and send the partial bitstream to the FPGA directly via the secure connection to the bootstrapping bitstream. As shown in Fig. 4d, after a secure session is started with the infrastructure code, the client can upload their application. As shown in Fig. 5, the infrastructure core can receive this partial bitstream, decrypt and authenticate the binary with the session key and send it to the ICAP for partial reconfiguration (checking to ensure that the bitstream only attempts to program logic within the appropriate partial reconfiguration region).

However, this added flexibility also adds system complexity and could introduce security vulnerabilities (e.g. the ICAP is a bridge between "program" space and "data" space). Operations such as tokenization appear in many different applications and will likely be used by many different clients. Such common operations can be offered as specific secure computation service by the cloud provider. As will be discussed in Section 7, for the moment our proof-of-concept system only implements this "virtual HSM" mode of operation. However, with the exception of the ICAP and partial bitstream verification logic, all of the infrastructure is in place to provide more extensible functionality.

Regardless as to whether the client directly develops their secure computing application or merely chooses among pre-made applications, the bitstreams representing these computations are encrypted. Thus, the bitstreams not only define computation, using initial values they can store pre-set keys. Just as this worked for *privatek$_f$*, to define the device's identity, it can also work for any application-specific keys. Thus, the application developer could define a static key for encrypting and tokenizing data, separate from the session key clients use to communicate with the system.

Returning to our example, after the client establishes a secure session to a device that supports their computation, they can send sensitive input data from their local machine (encrypting with the session key *sessionk$_f$*.) and either receive output data back in their local machine or on cloud machines. This is shown in Fig. 4e.

5.3. System Updates

It may be necessary to perform updates to the system. Any issues that are found in computations performed in programmable logic are relatively easy to fix. These include:

a) The PKI operations for session key negotiation are compromised (i.e. new vulnerabilities are found in RSA or SHA).
b) The private RSA key *privatek$_f$* is lost (i.e. others may be able to spoof the public identity of the trusted node).

In this case, the trusted authority simply needs to generate a new encrypted bootstrapping bitstream – the same operation needed to supporting new virtual HSM applications.

However, this system does rely on certain immutable logic pieces and data that is comparatively difficult to update. For example, if a new vulnerability were found in AES, this would not be an issue for the user application in Fig. 5. The client application could simply change to a different encryption algorithm or a different key length.

On the other hand, this would be a serious problem for the system as a whole. This is because the immutable bootstrapping logic forms the initial "root of trust". AES is implemented in static gates in the bootstrapping logic rather than programmable logic. If a vulnerability is found in AES, the FPGA must be replaced to replace the bootstrapping logic.

Similarly, *symk*$_{fb}$ is also an essential part of the initial root of trust. If the key for a given device were somehow leaked from the TA, attackers could reverse-engineer the encrypted bootstrap bitstream and retrieve *privatek*$_f$.

Updating this key is difficult because existing FPGAs have no internal connection to overwrite the key memory from the logic fabric. The external configuration memory plus internal key memory arrangement used in most FPGAs has only been well vetted by manufacturers for the purpose of IP security. Because of this, the manufacturers never intended for the contents of the onboard key memory to be updated in the field. The only way to update the key is via an external connection, so replacing the key requires a secure setting. For example, the board would most likely need to be returned to the TA for re-provisioning.

Looking ahead, FPGA manufacturers can mitigate the issues regarding algorithmic vulnerabilities in the immutable bootstrapping logic. Existing bitstream encryption is designed for IP protection for general systems. Many of these general applications may need very fast initialization of the device to meet system-wide power-on timing requirements (e.g. BIOS device enumeration). Modern FPGAs can boot from flash in less than a second. However, in our use case, the infrastructure bitstream may be booted once, and the system may be on for quite some time afterwards. In this case, fast booting is not particularly important. On the other hand, any vulnerability in the encryption used to build the original root of trust would be catastrophic. For these reasons, it would make sense for FPGA manufacturers to offer higher security, but potentially slower algorithms. For example, a larger bitwidth key, additional rounds, or even multiple different encryption algorithms could be implemented.

Similarly, FPGA manufacturers can easily mitigate the issues regarding updating the internal key memory. This can be accomplished by simply adding a write connection port from the programmable fabric to the key memory. Although this may allow "rogue" bitstreams to wipe out the key, effectively bricking the device, this is no worse than can be done via the external write port.

6. EXPERIMENTAL RESULTS

We prototyped our proof-of-concept system using the Xilinx ML605 board containing a Virtex-6 LX 240T. Logic and memory utilization of our system is shown in Table 1.

Our infrastructure and tokenization logic is largely comprised of various cryptography circuits. Prior work has shown that these types of operations can be very efficiently mapped to FPGAs [8, 9, 10]. In this regard, the performance and resource requirements of our proof-of-concept system are no surprise.

The amount of resources required is relatively small. The infrastructure logic (PCIe & DDR3 controllers, and key exchange logic) only require 14.8% of the available LUTs, 8.6% of the available FF, 5.2% of the available block RAMs and 0.5% of the available DSP multipliers. Most of these resources are consumed by the PCIe and DDR3 controllers.

The infrastructure logic is also very fast. Our 2048-bit RSA implementation uses a 200 MHz clock and can perform ~13 operations per second in the worst case. It takes 76ms to perform a session key exchange.

The tokenization is also very small and efficient. Tokenization with two AES cores and a SHA-256 core only requires 3.3% of the available LUTs, 0.3% of the available FFs, and 0.7% of the available block memories.

Again running with a 200MHz clock, our 256-bit AES implementation can process ~572MB/s in sustained operation. This easily outstrips the bandwidth of the Gigabit Ethernet link on the host server. Our SHA-256 implementation can process at ~12MB/s. For our anticipated use-case, this also easily outstrips the bandwidth of the host's network link. This is because only PII is tokenized. Even a single SHA core will be sufficient to service the needs of a Gigabit Ethernet link if less than 10% of the data coming into the system is PII. Adding a second SHA core will require very little additional resources and will allow the system to handle the needs of a Gigabit Ethernet link if less than 20% of the data coming into the system is PII. In the databases considered for this work, between 1% and 10% of the information was PII.

Table 1. Resource Utilization (V6 LX240T).

	LUTs	*FF*	*BRAM*	*DSP*
Full System	27237 (18.1%)	27076 (9.0%)	49 (6.9%)	4 (0.5%)
· Infrastructure	22314 *(14.8%)*	26030 *(8.6%)*	43 *(5.2%)*	4 (0.5%)
RSA	*2149 (1.4%)*	*2107 (0.7%)*	*3 (0.4%)*	*4 (0.5%)*
SHA-256	*814 (0.5%)*	*697 (0.2%)*	*2 (0.2%)*	*0 (0.0%)*
PCIe Controller	*13499 (9.0%)*	*14910 (5.0%)*	*38 (4.6%)*	*0 (0.0%)*
DDR3 Controller	*5556 (3.7%)*	*8092 (2.7%)*	*0 (0.00%)*	*0 (0.0%)*
· Tokenization	4923 *(3.3%)*	1046 *(0.3%)*	6 *(0.7%)*	0 *(0.0%)*
AES x2	*3662 (2.4%)*	*272 (0.1%)*	*4 (0.5%)*	*0 (0.0%)*
SHA-256	*814 (0.5%)*	*697 (0.2%)*	*2 (0.2%)*	*0 (0.0%)*

7. FUTURE WORK

Looking ahead, there are two aspects of the system we would like to investigate further. First, we would like to look for additional applications that can benefit from the scale of the cloud, but cannot migrate due to security concerns. These include areas such as secure database operations and handling payment card transactions.

Second, as alluded to in Section 5.2, there are both advantages and disadvantages of migrating from our current compute model, where we offer secure pre-compiled "application services", to a more flexible one, where clients can define their own applications without involving the trusted authority directly.

One problem that we currently sidestep is the engineering effort of developing hardware applications, particularly by cloud clients. It may be unrealistic to assume that most cloud customers will be willing to code their applications in Verilog or VHDL. This makes interfacing the system with a high-level language compiler a necessity. We would like to investigate the different ways this integration might be performed and the practical limitations of the various high-level language toolflows.

Another set of issues we have not addressed are the concerns that arise when we allow dynamic reconfiguration. Although there are certainly sufficient logic and memory resources on the device to implement an ICAP, this has non-trivial security implications. The frame-based nature of the Xilinx configuration bitstream gives us some way of ensuring safe dynamic reconfiguration (i.e. not allowing the reconfiguration of frames outside of the dynamic region). However, by itself this may not be sufficient. For example, we would like to make the dynamic region as large as possible to make best use of the available hardware. That said, many resources such as I/O pins and hard macros have fixed locations. This likely impacts the high-level design.

Similarly, route-though wires often need to cross a dynamic region to meet timing. Here, we would like to investigate two issues. First, looking at the possibility of performing on-board "deep" inspection of dynamic bitstreams to verify that new applications do not eavesdrop on or otherwise alter route-though wires. Second, looking at ways of partitioning the infrastructure logic with regards to signals that do not need to be held securely (and thus can safely route-through a dynamic region even in the face of potential tampering) and those that to need to be held securely (and thus should not route through a dynamic region if at all possible).

8. CONCLUSION

Cloud computing presents a unique security challenge. In contrast to traditional private servers, client may not trust system administrators or the integrity of the machines themselves. Since many applications only perform a small amount of processing on sensitive data, we can address these concerns by offloading these operations to a trusted computing platform. However, given the new security demands placed on cloud systems, we may not be able to build these trusted computing nodes effectively using traditional CPU-based systems.

In this paper, we introduce the idea of using FPGAs to build a flexible trusted computing platform. Hardware-based systems can solve the issues that affect traditional software-based systems by providing a well-defined and significantly smaller attack surface. This allows us to offer stronger guarantees that are more robust against attack.

Although true homomorphic encryption may be years away for real-world use, FPGAs offer a unique practical alternative, emulating the effective behavior.

9. REFERENCES

[1] V. Vaikuntanathan, "Computing blindfolded: New developments in Fully Homomorphic Encryption." IEEE Foundations on Computer Science, 2011.

[2] Trusted Computing Group, "TPM Main Specification Level 2", Version 1.2, Revision 116. http://www.trustedcomputinggroup.org/resources/tpm_main_specification

[3] Xilinx Corporation, "Virtex-6 FPGA Configuration" User Guide UG360, v3.4, 2011

[4] T. Eisenbarth, T. Guneysu, C. Parr, A. Sadeghi, D. Schellenkens, and M. Wolf, "Reconfigurable Trusted Computing in Hardware," ACM Conference on Computer and Communications Security, 2007.

[5] C. Gentry. A fully homomorphic encryption scheme. PhD thesis, Stanford University, 2009.

[6] L. Bouganim and P Pucheral, "Chip-Secured Data Access: Confidental Data on Untrusted Servers," VLDB 2002.

[7] IBM Corporation, "IBM PCIe Cryptographic Coprocessor". http://www-03.ibm.com/security/cryptocards/pciecc/overhardware.shtml

[8] A. Elbirt, W. Yip, B. Chetwynd and C.Parr, "An FPGA-based performance evaluation of the AES block cipher cadidate algorithm finalists," IEEE Transactions on VLSI Systems 9(4), 2001

[9] T. Blum, C. Paar, "High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware", IEEE Transaction on Computers 50(7), 2001.

[10] K. Ting, S. Yuen, K. H. Lee, P. Leong, "An FPGA Based SHA-256 Processor", International Conference on Field Programmable Logic and Applications, 2002.

[11] Thales Corportation, "Hardware Security Modules". http://www.thales-esecurity.com/EN/Products/Hardware%20Security%20Modules.asp