Process Isolation for Reconfigurable Hardware

Herwin Chan
Electrical Engineering Department
University of California, Los Angeles
Los Angeles, CA
herwin@ee.ucla.edu

Patrick Schaumont
ECE Department
Virginia Tech
Blacksburg, VT
schaum@vt.edu

Ingrid Verbauwhede
ESAT / SCD-COSIC Department
Katholieke Universiteit Leuven
Leuven, BE
Ingrid@ee.ucla.edu

Abstract

One of the pillars of trust-worthy computing is process isolation, the ability to keep process data private from other processes running on the same device. While embedded operating systems provide isolation for the software part of these processes, there is no commonly accepted isolation mechanism for the hardware resources. As a result, systems may remain vulnerable to hardware-based attacks. This paper presents a secure coprocessor interface that extends the concept of process isolation into reconfigurable hardware. In the resulting coprocessor design, context information for different processes concurrently accessing the coprocessor is physically kept private. The coprocessor interface can handle context switches between different processes without assistance of the operating system. Because of this, reconfiguration of computation units can occur independent of the main processor. Moreover, it does so with greater efficiency than what is possible using software only.

1. Introduction

The internet is dominant for transferring information between machines and it is becoming increasingly important to protect this data. The AES algorithm [1] is the new standard in symmetrical cryptography and has been the subject of hardware acceleration. Developments in this area focused on such design goals as fastest design [2], most efficient design [3], or the most flexible design [4,5]. However, none of these schemes addresses the issue of security between the different processes sharing a single coprocessor.

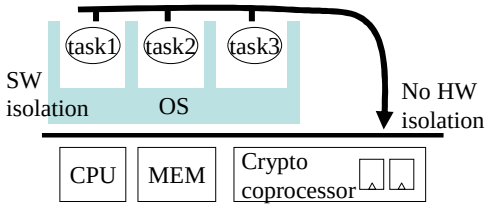

Figure 1. Lack of hardware isolation in current architectures

In this paper, the terms task and process are used interchangeably. Traditionally, these two software entities differ in the amount of context information that is considered private. Since our coprocessor can secure context information for any software entity, the two terms are indistinguishable to the coprocessor.

On a server, there may be many different independent secure connections alive at any moment. In current implementations, a separate software process using a shared coprocessor for cryptographic acceleration, handles each connection. As Figure 1 shows, even though a secure operating system can enforce isolation, the coprocessor still can leak information between independent processes through the coprocessor registers.

Multitasking operating system security focuses on the implementation of two main goals: resource access control and resource isolation [6]. Resource access control is the assignment of permissions to use system resources. The classic trusted operating system heavily relies on the access control strategy by explicitly assigning access rights to processes and controlling the interaction between users and various system objects (such as files, IPC, and the network stack). Resource isolation ensures that data from one process is not able to leak to another process. This is the strategy of UNIX processes where each instance of an application runs in its own virtual address space.

Though access to the coprocessor can be managed in the OS, process isolation remains a problem. This paper describes a coprocessor interface that addresses this issue. Direct application of the results of this work increases security by complementing and enhancing existing security design techniques.

Security in computing systems is often associated with the notion of trusted computing. Work in this area mainly focuses on mechanisms to ensure that only trusted software is allowed to be installed or executed. Though well promoted by industry, the effectiveness of such an approach is debatable [7,8]. In this paper, we focus on providing security by process isolation though a secure channel; malicious programs may run on our system, but they cannot interfere with or snoop on other processes. The idea of using hardware to provide

isolation has been explored in several related works. Trusted Logic [10] offers an operating system framework that isolates security services from their environment. A secure hardware channel is established so that application software can communicate with software security services. In our work, security services are implemented in hardware and the secure communication channel is established by the coprocessor interface.

The security architecture of the CELL Broadband Engine [11] processor is able load a program onto one of its Synergistic Processor Element (SPE) cores and run it in isolation mode. In this mode, neither the executing program or its data can be observed or manipulated. This framework, however, limits the interactivity of the isolated program with other processes in the system. Our scheme provides a secure hardware interface where interactive services can be easily accessed by other software processes.

We also differentiate our system from FPGA security in general, which focuses on protecting the bitstream that describes the system's hardware configuration. Our system protects the data from processes in the system and not the actual hardware.

Processor / coprocessor communication overheads are a large source of system inefficiency. Overheads include coprocessor reconfiguration time and transmission of context information before a calculation can occur. Each process that needs cryptographic services has its own secret key, initial vectors, and modes of operation. This problem severely limits the AES core from achieving its full potential, and is most noticeable in common internet communications where short bursty data packets dominate [9].

This paper describes a new type of coprocessor interface that addresses the issues of security and performance in a reconfigurable platform. It provides a mechanism to ensure security between software processes (some of which may be malicious) and to minimize the overhead associated with context switching between multiple processes and hardware reconfiguration. In addition, since these features do not require any OS support, existing systems can incorporate them easily. The processor described in this paper uses a generic AES core but the scheme is applicable to any other type of crypto-coprocessor.

Section 2 describes the function of the coprocessor and its place in common system architectures. To illustrate problems with traditional interfaces, we give some motivating examples. Section 3 describes the architecture of the AES secure multitasking coprocessor. Section 4 presents the results of synthesis and system co-simulation. Finally, Section 5 concludes with the main ideas.

2. System Architecture

Figure 2 shows three possible system architectures for an AES coprocessor. The dashed lines represent the flow of data and the solid lines represent the flow of control signals. In Figure 2a, the coprocessor directly connects to the microprocessor. This means that both data and control signals must pass through the microprocessor, making this the main bottleneck in an AES operation. In the second architecture (Figure 2b), the coprocessor connects directly to a streaming interface that will supply the data. With this architecture, the microprocessor only deals with control of the coprocessor and interactions proceed on a block-by-block basis rather than a word-by-word basis.

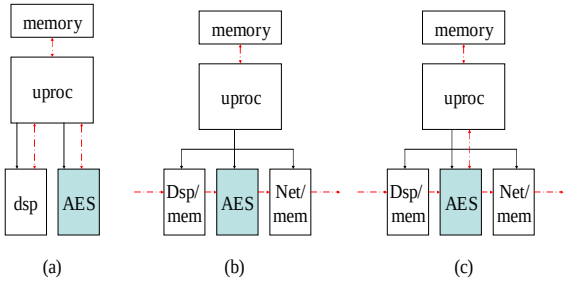

Figure 2. Different system architectures: a) simple coprocessor interface, b) streaming coprocessor interface, and c) hybrid coprocessor interface

The most significant source of communications overhead is the transfer of context. Cryptographic context is the process specific data such as the secret key, the mode of operation, and initial values. The transfer of this information for each process not only reduces system efficiency, but also increases the risk of data interception.

Architecture 2a may seem to incur a large overhead but has the benefit of a more traditional architecture and is more appropriate for interactive type applications such as a telnet session. Architecture 2b is more appropriate for processing of continuous streaming data or large data blocks. To maintain architectural flexibility, the multitasking AES processor simultaneously supports both of these architectures (Figure 2c).

Allowing reconfiguration of the AES coprocessor can also increase system processing overhead. Though the processing components within the coprocessor can dynamically change to improve system performance, this usually requires the suspension of all calculations while reconfiguration takes place. Our system avoids this obstruction by modular design of the coprocessor architecture and implementation of a transaction based interface.

We now present two motivating examples that illustrate the purpose of our multitasking coprocessor interface. The examples illustrate security and context switching operations.

2.1 Security Example

One possible attack can occur at the moment that a process has just used the coprocessor to encrypt a command to a remote server. A malicious process may then use the same coprocessor without reprogramming the settings to send its own data to the remote server, essentially spoofing the identity for the first process.

In a related attack, the malicious process can partially reprogram the coprocessor by just changing modes from encryption to decryption. The previous encoded output can then be reinserted into the coprocessor to reveal the unencrypted message.

With current coprocessors, this can be prevented by having the tasks reset the coprocessor after each operation. However, this solution imposes additional overhead, because it increases the period within which the coprocessor interface will remain locked by a single task. Moreover, it leaves the responsibility of security on the design of the operating system. Operating systems themselves are very complicated software structures.

The idea of the multitasking AES coprocessor is to have a hardware solution to this problem. The cryptographic state of a process will be securely stored and managed on the coprocessor itself. To guarantee security of the state information, a separate dedicated controller manages the state of each process.

2.2 Context Switch Example

Assume that a single AES coprocessor encrypts two channels of streaming data. Each channel has a different mode of operation and different keys. The streaming data is time sensitive and must have its latency bounded by a certain value. For a traditional interface, this would mean that the operating system would have to manage the context switching between these two tasks. The overhead due to software context switching and repeated interactions with the coprocessor is a limitation on the total throughput of the system.

A system with many processes having bursty data illustrates an extreme example of context switching. This situation is not unusual in a VPN server, which handles large number of secure interactive sessions.

The multitasking AES coprocessor stores the contexts (process specific data such as the key and mode of operation) of the processes in the coprocessor. This means that for streaming applications, the bandwidth of the AES core is shared among the active tasks with no time lost to context switching. For extremely bursty traffic, the context information is already stored on chip; therefore, overheads associated with processor-coprocessor interactions are minimized.

3. AES Coprocessor

3.1 AES Algorithm

The AES cipher is a block cipher [1], which means that encryption and decryption operate only on fixed blocks of data. In our implementation, 128 bit is the block size.

The algorithm consists of five main operators: AddRoundKey, SubBytes, ShiftRows, MixColumns, and KeyExpansion. The inverse of these operators are used for decryption. Figure 3 shows the how these operators are used to perform encryption and decryption.

Encryption	**Decryption**
AddRoundKey	AddRoundKey
For round = 1 to 9	For round = 1 to 9
SubBytes	InvShiftRows
ShiftRows	InvSubBytes
MixColumns	AddRoundKey
AddRoundKey	InvMixColumns
SubBytes	InvShiftRows
ShiftRows	InvSubBytes
AddRoundKey	AddRoundKey

Figure 3: Pseudocode for AES encryption and decryption

In both encryption and decryption algorithms, there is a FOR loop that runs through four of these steps. Hardware is efficiently realized by implementing only this group of operations (a single round) into hardware. The same hardware can then be used several times to perform a single encryption or decryption operation.

3.2 Coprocessor Architecture

Figure 4 shows the main logical blocks in the coprocessor. The processor interface block accepts instructions from the microprocessor through a memory-mapped interface. This block will then assign the work to one of the agent blocks. The agent blocks are dedicated controllers that are able to perform AES encryption and decryption in any mode of operation. The agent blocks each have enough registers to store the state of the calculation. The AES core performs the actual calculations. In our implementation example, this is a purely combinational block, which performs a single round of encryption or decryption; eleven rounds are necessary to perform a single AES calculation.

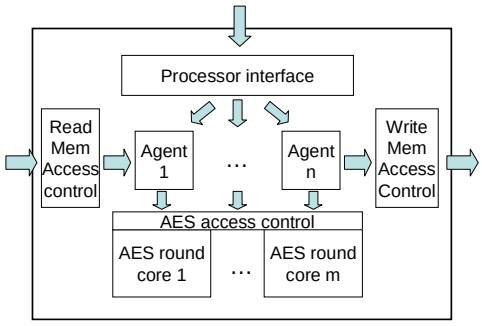

Figure 4. Architecture of multitasking coprocessor

There are several agents in the coprocessor managing multiple AES cores (each responsible for calculating a single round). The number of these elements change depending on the throughput and latency requirements of the system. A round robin scheduling algorithm is used to ensure fair access to the AES cores. Memory read and write access control blocks are available to support the streaming or block processing architecture of Figure 2b. For easy integration with popular architectures, all interfaces are 32-bit buses.

Both encryption and decryption support the following modes of operation: electronic codebook (ECB), cipher block chaining (CBC), cipher feedback (CFB), output feedback (OFB), and counter (CTR). Such flexibility enables support of a wide variety of popular security protocols including IPSec, SSH, and SSL/TLS.

The following subsections explain the detailed functions of the main blocks in our coprocessor architectures. This includes the processor interface, agent blocks, and access control.

3.3 Processor Interface

The microprocessor connects to the coprocessor through a memory-mapped interface. The processor interface uses an instruction set designed to minimize the amount of communications necessary. Figure 5 shows the format of the instructions the coprocessor receives. Depending on the type of command, zero or more of the optional fields are used.

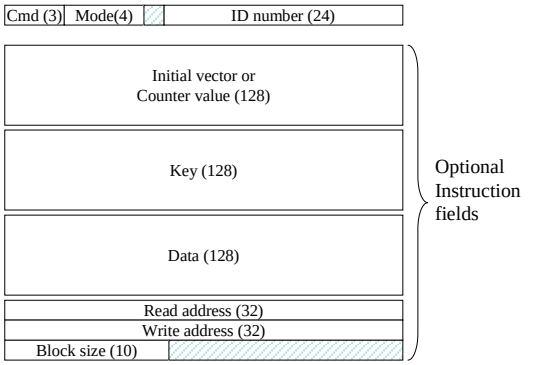

Figure 5. Instruction format of the coprocessor

The normal use of the protocol is shown in Figure 6 and proceeds in the following manner: A process sends a command to the coprocessor to reserve some resources for future calculations. If resources are available, the coprocessor grants the request by returning a random and unique ID number. Future commands will use this ID number to reference the cryptographic context in which calculations occur. At the end of a process' life, the processor gives the command to the coprocessor to release the reserved resources.

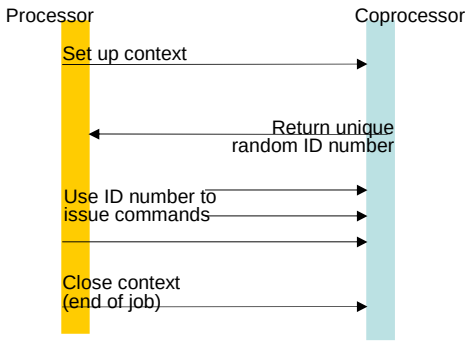

Figure 6. Processor / coprocessor interface protocol

It should be noted that the protocol allows a software process to identify it's assigned agent through the random ID number. However, the reverse is not true. The hardware agent cannot identify its corresponding process and implicitly trust all processes based on the ID number.

The protocol also allows using the coprocessor without the reservation of resources. In this case, the coprocessor will return the result upon the completion of the calculation.

Table 1 shows the five types of commands that are available and the amount of communications needed to complete them.

TABLE 1. Commands accepted by the coprocessor

Cmd	Description	Return value	Words rd+wr
Agent setup	Reserve and configure an agent for AES calculation	ID number or FALSE	9+1
Agent check	Check to see if previous calculation has been completed	DONE or FALSE	1+1
Agent release	Clear a certain context from the coprocessor	DONE or FALSE	1+1
Agent single	Perform a single preset AES calculation	AES result or FALSE	5+4
Agent continuous	Perform a series of preset AES calculations taking data directly from memory	DONE or FALSE	4+1
Immediate	Perform a single AES calculation	AES result or FALSE	13+4

3.4 Agent Blocks

The architecture of the agent block is shown in Figure 7. These blocks are responsible for managing calculations for a single process. The task given to an agent may be to encrypt a large data file. In this case, the agent block retrieves the data from memory, performs multiple AES calculations, and then writes the encrypted data back to memory. Because there is a tight coupling between a software process and it's agent, many agents blocks exist within the multitasking coprocessor.

The agent consists of a small finite state machine with a collection of registers to remember the state of AES calculations.

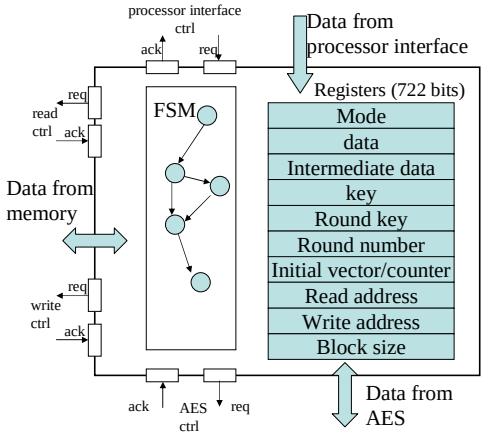

Figure 7. Architecture of the agent blocks

The values stored in the registers add up to 722 bits of data and contribute to over 60% of the area of this block. In designs where many agents are required, area can be saved by using an AES core that performs a full AES calculation; intermediate data storage (which accounts for 30% of the total registers) will not be required in the agent. The result is a larger AES core and a system with slightly longer latencies. Instead of registers, a common RAM module for all the agents can also achieve a more area efficient but lower performance design.

3.5 Access Control

Access control blocks regulate admission to the AES core and the external memory blocks by the agents. Each of these blocks implements a round robin priority scheduler. This means that the priority of the agents to use the resources rotates each clock cycle. This ensures fairness among the agents competing to use the resources and guarantees that all calculations experience the same latency.

3.6 Reconfiguration

The interface protocol introduced in subsection 3.3 is transaction based. It serves to isolate the requests made by the main processor from the active components performing the computation. In a reconfigurable system, this allows the number of agents and AES cores to change without the knowledge of the application software. This results in smaller and more portable software (multiple versions for each dynamic configuration is no longer necessary).

The AES coprocessor can be configured by specification of the number of agent blocks and associated AES cores. The two parameters, the number of agents and the number of AES cores, affect the performance of the system differently. The number of agent blocks determine the number of simultaneous processes that may be handled. The number of AES cores determine the maximum throughput which the system can support. For an efficient system, these two parameters are determined based on the required throughput and latency of the processes.

The agents and AES cores all operate independently from each other. The agents are isolated from the main processor through the processor interface block. The AES cores are isolated from the agents through the access control block. Because of this modular architecture, it is possible to add or remove elements dynamically without halting currently running processes. This further increases the efficiency of the platform.

4. Results

We implemented and tested the coprocessor design in our system design environment. The following sections analyze the resulting performance and cost of the system.

4.1 Design Size and Speed

To examine the relative size of each of the modules in the design, we synthesized the design for the Virtex-II Pro FPGA using Synplicity. Table 2 shows the results.

TABLE 2. Size and speed of modules in the coprocessor

Module	Slices	Critical path (ns)
AES core	3037	--
AES access controller	132	2.9
Agent (each unit)	1065	7.6
Read memory access controller	186	6.2
Write memory access controller	12	1.9
Microprocessor interface	623	5.8

Agents take up about a third the area of the AES core. This suggests that system performance can be easily increased by adding agents and increasing system efficiency. The resulting area/performance ratio will be lower than if only AES cores are added to the coprocessor.

4.2 Performance Analysis

In order to show the benefits of our coprocessor design at the system level, we analyze its performance using real world data. Studies such as [12] shows that 90% of internet traffic is under 1Kbytes. In our test scenario, a packet size of 1Kbytes is assumed. We can then measure the time it takes to process each of these packets. Table 3 shows the results of the comparison.

TABLE 3. Comparison of overhead for bursty traffic loads

	Context switch (cycles)	AES calc (cycles)	Total Time (cycles)	efficie ncy
Multitasking	54	63	117	54 %
Traditional	194	63	257	25 %

This result shows that the multitasking interface can handle more than twice the number of 1Kbyte packets as the traditional coprocessor interface. However, the actual AES core is still running at half its capacity. This suggests that further improvements are possible in the instruction set design of this type of coprocessor. Future versions should further optimize the design to increase the capacity for the bursty traffic model.

The coprocessor is also able to encrypt several continuous data streams. This traffic pattern is common for multimedia type applications. Latency is often important in teleconferencing applications and they exhibit this type of traffic pattern. Table 4 shows how the latency changes as the coprocessor processes multiple data streams.

TABLE 4. Comparison of latency for different number of simultaneous streams

Number of simultaneous streams	Multitasking interface latency (clock cycles)	Traditional interface latency (clock cycles)
1	22	12
2	28	400
3	36	594
4	48	788

For a single stream, the traditional approach outperforms the new multitasking interface approach. However, for multiple data streams the multitasking coprocessor is able to scale much more gradually. In our interface, context switching is performed in hardware at the AES round level. Consequently, the latency increases much more gradually.

This effect becomes much more serious for traditional interfaces when implemented in a network that processes both bursty packets and continuous streams. The high frequency of bursts can severely degrade the latency of the stream processes.

Note that agents are hardware objects designed to make efficient use of the computational resource, in this case, the computation of one AES round. The overall throughput of the system, however, is limited by the AES core. For systems requiring increased throughput, multiple cores must be created.

4.3 Application Profile

A secure data server application was implemented on top of multithreaded software simulation platform. When a client establishes a connection with the server, a key and encryption mode of operation is negotiated. Data is then encrypted and sent to the client. Several clients can be handled simultaneously and a process is created to manage each connection. The footprint of the different software components is shown in Table 5.

TABLE 5. Size of software components

Server Application	2,794 bytes
SW AES	33,536 bytes
Coprocessor interface drivers	2,928 bytes
Quickthreads library	1,868 bytes
Multithreaded TCP/IP communications stack	106,957 bytes
System calls	4,508 bytes
TOTAL	152,591 bytes

From the system point of view, the size of the software can be reduced by 20% if a coprocessor is used to perform AES encryption. This shows that for a data

server application, it is possible to reduce cost, increase performance and increase security together with our proposed coprocessor.

5. Conclusions

Though security in multitasking systems is traditionally the domain of the operating system, hardware solutions can offer similar protections. This paper describes a coprocessor interface for crypto-processors that prevents the access or use of secret information from software processes running on a common processor.

Conventional coprocessor interfaces do not offer any features to protect data in a multitasking environment. In addition, because of the high context switch times, overall system throughput is degraded under bursty traffic loads.

The coprocessor interface described in this paper address both these issues. The use of small-distributed agents in the coprocessor physically separates data from different software processes. By assigning unique and random ID numbers to agents, software processes running on the microprocessor are unable to access data from one another.

We demonstrate that secure coprocessors do not need the support of a large and complicated software infrastructure. Because of this independence, the coprocessor interface can be added to existing systems with minimal design overhead.

Traditional solutions to multitasking security explicitly define resources to which a process has access. In this work, services are not denied if resources are available. Instead, security is created through the protection of cryptographic contexts that exist in the system. A software process binds to its context in the coprocessor through the 24-bit ID number assigned at time of creation.

In addition to greater security, performance is also improved. There is tight coupling between the access control blocks and the agents. This allows efficient sharing of the AES cores so that the maximum throughput is maintained even during multiple simultaneous executions of calculations having different modes of operation. The access control blocks also serve to isolate the processing elements. This allows reconfiguration of the coprocessor without suspension of calculations in progress.

The concepts described can be adapted to apply to any previously designed AES core and directly adds the security and multitasking features necessary in real systems. The number of agents in the coprocessor is easily adjustable at design time to tune the performance for particular expected traffic patterns.

Acknowledgment

This material is based upon work supported by the Space and Naval Warfare Systems Center - San Diego under contract No.N66001-02-1-8938, NSF (Grant CCR-0310527), and SRC (Grant SRC-2003-HJ-1116).

References

[1] National Institute of Standards and Technology (U.S.), Advanced Encryption Standard. http://csrc.nist.gov/publication/drafts/dfips-AES.pdf

[2] A. Hodjat and I. Verbauwhede, “Minimum area cost for a 30 to 70 Gbits/s AES processor,” Proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI ’04), pp. 498-502, February 2004.

[3] C.-P. Su, T.-F. Lin, C.-T. Huang, and C.-W. Wu, “A Highly Efficient AES Cipher Chip,” Proc. of Asia and South Pacific Design Automation Conference ASP-DAC 2003, pp.561-562, January 2003.

[4] F.K. Guurkaynak, A. Burg, N. Felber, W. Fichtner, D. Gasser, F. Hug, and H.Kaeslin, “A 2 Gb/s Balanced AES Crypto-Chip Implementation,” Proc. of the 14th ACM Great Lakes Symposium on VLSI, pp.39-44, 2004.

[5] M. McLoone and J.V. McCanny, “Generic architecture and semiconductor intellectual property cores for advance encryption standard cryptography,” Proc. IEE Computers and Digital Techniques, July 2003.

[6] P-H Kamp and R. Watson, “Building Systems to Be Shared, Securely,” ACM Queue, vol. 2, issue 5, pp. 42-51, July/August 2004.

[7] S.J. Vaughan-Nichols, “How trustworthy is trusted computing?” Computer, vol. 35, issue 3, pp.18-20, March 2003.

[8] R. Oppliger and R. Rytz, “Does trusted computing remedy computer security problems?” IEEE Security and Privacy Magazine, vol.3, issue 2, pp.16-19, March/April 2005.

[9] D. Whiting, B. Schneier, and S. Bellovin, “AES Key Agility Issues in High-Speed IPsec Implementations”, May 2000. http://www.schneier.com/paper-aes-agility.html

[10] http://www.trusted-logic.com/mob_tech.html

[11] http://www-128.ibm.com/developerworks/power/libaray/pa-cellsecurity

[12] N. Brownlee and K. Claffy, “Internet stream size distributions,” Proc. of 2002 ACM SIGMETRICS international conference on Measurement and Modeling of Computer Systems, pp.282-283, June 2002.