Application of Cryptographic Primitives to Computer Architecture

Ted Huffmire
Computer Architecture Laboratory
University of California, Santa Barbara
huffmire@cs.ucsb.edu
March 2, 2005

Abstract

Cryptography and computer architecture have co-evolved since the beginning of computer science. For computer architects, security has become a first-class design constraint alongside power and performance. Computer architects often rely on heuristics that only work well probabilistically, and cryptographic primitives such as incremental hashing can provide guarantees on the accuracy of these heuristics in the worst case. This paper will cover the state-of-the-art in cryptographic processors, which can encrypt or decrypt a buffer rapidly, and secure architectures, which use crypto in a smart way to prevent piracy and tampering. I will also briefly discuss my research applying incremental hashing to provide cryptographic guarantees on the accuracy of dynamic memory allocation analysis of computer programs.

I. Introduction

I. A. Why Crypto is Important to Computer Architects

Cryptography is important to computer architects because crypto is essential to computing. For example, secure routing protocols require high bandwidth encryption. One important goal of computer architecture (CA) is to make computers run fast by finding performance bottlenecks and improving efficiency by understanding the basic building blocks that are common to a group of problems. However, computer architects cannot operate in a vacuum, assuming that the operating system or application software will take care of the security problem. On the contrary, security must become integral to the design of a processor. CA is a cross cutting discipline that not only can help other areas of computer science improve their performance but also aid in the security battle.

I. B. How CA is Useful to High-Speed Crypto

CA is useful to cryptography because hardware implementations provide speed and physical security. For example, wireless devices are exposed to a hostile environment, and their embedded processors must be able to encrypt and decrypt efficiently without wasting power. For years, many computer scientists have assumed that the radio burns the most power in a wireless device. However, researchers have recently identified a “security processing gap” for wireless devices [Ravi 02]. For example, the Palm III requires 3.4 minutes to generate a 512-bit RSA key, 7 seconds to generate a digital signature, and it can only perform DES encryption at a rate of 13 kbps [Ravi 02]. Fast crypto can also help the high-speed networking community. For example, Secure IP and Virtual Private Networks require high bandwidth encryption.

I. C. The Need for Secure Architectures

Embedded processors are part of everyday appliances from televisions to automobile navigation systems, and computer architects must consider the susceptibility of their processor designs to tampering. For example, a virus can infect the navigation system of certain models of Lexus vehicles via the Bluetooth mobile phone connection [Quainton 05]. By applying crypto to the processor design in a smart way, it is possible to curtail tampering and piracy.

I. D. Crypto in a Nutshell

There are three kinds of cryptography: asymmetric (a.k.a. “public-key”), symmetric (a.k.a. “private-key”), and one-way hash. Ciphers are designed so that an attacker must solve a very difficult mathematical problem to break the encryption. In asymmetric cryptography, each party has a pair of keys, one public key and one private key. Users must keep their private keys secret, and they may publish their public keys. Suppose that Alice wants to send a message to Bob. She will encrypt the
message (a.k.a. “plaintext”) using Bob’s public key and send the resulting ciphertext to Bob. Bob can decrypt the message with his private key. In symmetric cryptography, each party encrypts and decrypts with the same private key, which must be exchanged ahead of time using a key exchange algorithm such as Diffie-Hellman. One-way hash functions transform a variable length string into a string of fixed length, known as the hash. It must be very difficult to guess the original string given only the hash, and it must also be very difficult to find two strings that result in the same hash.

Although asymmetric encryption is convenient because a key does not need to be exchanged ahead of time, it is much more expensive than symmetric encryption. Therefore, asymmetric encryption is often used to exchange a symmetric “session key.” Another example of combining different kinds of cryptography together is digital signatures, which combine asymmetric cryptography and one-way hash functions. Suppose Alice has authored a document and wants to digitally sign the document so that Bob knows that Alice wrote it. Alice generates a one-way hash of the document and then signs the hash with her private key. When Bob receives the message, he decrypts the hash using Alice’s public key. Then, Bob also generates a hash of the document. If the two hash values are identical, then Bob has successfully verified the authenticity of the document.

I. E. The Need for Flexibility

Cryptography is a dynamic field, and ciphers are sometimes “broken.” In other words, some very clever person has found a mathematical technique to circumvent the encryption. The consequences of such a discovery are potentially serious for cryptographic hardware that has already been deployed. Redesigning a custom VLSI implementation or replacing hardware in remote systems is very expensive. Committees such as the Internet Engineering Task Force (IETF) routinely modify algorithm-independent protocols such as SSL and IPSec. Key lengths often need to be increased when more computing power become available to the adversary. All of these problems call for flexible hardware.

I. F. Field Programmable Gate Arrays

One example of flexible hardware is the Field Programmable Gate Array (FPGA). An FPGA is an array of reconfigurable blocks, and logic interconnects the blocks. FPGA implementations of algorithms are usually faster than software but slower than custom processors. FPGAs are much less expensive than custom VLSI fabrication, but they are not cheap enough for consumer products. The FPGA interconnect consumes 65% of total energy, but only 5% of the energy does any computation [Goodman 01]. General-purpose processors are much larger and consume more energy than FPGAs.

The remainder of this paper is organized as follows: Section II.A covers recent work in cryptographic processors, and Section II.B covers secure architectures. Section II.C introduces incremental hashing and its possible applications. Section III discusses my work applying incremental hashing to memory allocation analysis for computer programs. Section IV concludes and explains where there is room for future work.

II. The State-of-the-Art in Cryptographic Hardware

II. A. Cryptographic Processors

II. A. i. Goals of Cryptographic Processor Design

Cryptographic processors need to be able to encrypt or decrypt a buffer rapidly. They should support multiple ciphers, be inexpensive to manufacture, and use little power. It should be possible to upgrade crypto processors in the event that a cipher is broken, and a crypto processor should be scalable in case the key length of a cipher is increased.

II. A. ii. Energy Consumption of Crypto

Potlapally, Ravi, Raghunathan, and Jha measured the power consumed by an iPaq when executing different ciphers (Potlapally 03). They discovered that asymmetric crypto uses the most energy, followed by symmetric crypto, and then one-way hash functions. By varying the key length, they found that key length mostly affects asymmetric crypto power consumption. They argue that there is a tradeoff between the level of security and energy, and they suggest dynamic protocols that use weaker encryption when practical in order to save power.

II. A. iii. Symmetric Processors
Although asymmetric cryptography is more expensive than symmetric cryptography for equal session length, when the session length is long, the cost of symmetric crypto outweighs the cost of asymmetric crypto. CryptoManiac is a symmetric co-processor that supports multiple ciphers (Wu 01). Several ciphers were profiled to find bottlenecks. CryptoManiac is a 4-wide VLIW architecture with no cache or branch predictor because symmetric ciphers do not have branch or memory bottlenecks. The VLIW architecture executes a group of instructions at the same time. The host processor sends requests to a queue, and requests are dispatched to one of four CryptoManiac processors.

Cryptonite is another symmetric processor that is also capable of one-way hash functions [Oliva 03]. It has a two-cluster architecture so that key expansion can occur in parallel with encryption. Symmetric keys are much shorter than the length of the message, and they must be expanded before encryption or decryption can occur. Cryptonite has a reconfigurable permutation engine because permutations are fundamental to crypto and because different algorithms have different permutation tables.

SMP is another architecture that has been proposed for symmetric crypto [Dongara 03]. Most symmetric ciphers process data in blocks, and many use the Cipher Block Chaining (CBC) encryption mode, in which a plaintext block is XORed with the previous ciphertext block prior to encryption. Unfortunately, CBC is difficult to parallelize, which led to the development of Interleaved CBC (ICBC), in which multiple streams of CBC encryption are interleaved so that the encryption can be successfully parallelized on a SMP machine.

The three symmetric crypto processors presented are all flexible in that they all support multiple ciphers. They all have multiple processing elements that exploit the parallelism of symmetric crypto. CryptoManiac is the most flexible because it supports the most ciphers and has the most aggressive parallelization. The SMP design is slightly less flexible because it requires interleaving of the encryption streams. Cryptonite is the least flexible because it supports fewer ciphers and has less aggressive parallelization.

II. B. Secure Architectures

There is more to secure processors than simply encrypting a buffer. By using crypto hardware in an intelligent way, it is possible to prevent software piracy and system tampering. Theft of intellectual property has a significant impact on software companies. Anti-tampering technology is needed because memory errors caused by cosmic radiation can be exploited to take control of a virtual machine [Govindavajhala 03].

III. A. iv. Asymmetric Processors

RSA and Elliptic Curve Cryptography (ECC) are the two most popular types of asymmetric crypto. RSA is based on modular exponentiation, and ECC is based on point multiplication. ECC provides more security per bit of key length than RSA, a desirable feature in constrained environments. The processor proposed by Goodman and Chandrakasan can perform either RSA or ECC, and the computation unit is the only part of the processor that can be reconfigured dynamically, resulting in lower overhead [Goodman 01]. Leong and Leung propose an ECC processor with an ALU that performs field arithmetic rather than integer arithmetic [Leong 02]. Field arithmetic is the most basic building block of ECC, and it is needed to perform point multiplication. McIvor, McLoone, and McCanny propose an RSA processor with a Montgomery multiplier that can add very large operands without carry propagation [McIvor 03]. Modular multiplication is the most basic building block of RSA, and it is needed to perform modular exponentiation. The Montgomery Algorithm is one of the most efficient algorithms for performing modular multiplication.

The three asymmetric processors presented are all optimized to perform the basic operations of asymmetric crypto efficiently. Of the three designs, the processor proposed by Goodman and Chandrakasan is the most flexible because it can support either RSA or ECC. The design proposed by Leong and Leung is less flexible because it only supports ECC. The design of McIvor, McLoone, and McCanny is the least flexible because it only supports RSA, an algorithm that requires longer key lengths than ECC for the same amount of security. In general, symmetric crypto is more lightweight than asymmetric crypto, which explains why it is harder for asymmetric processors to support multiple algorithms.
Another example of tampering is monitoring energy consumption to extract cryptographic keys [Saputra 03].

Execute-Only Memory (XOM) is a proposal to prevent piracy and tampering [Lie 00]. In this scheme, the processor contains an asymmetric secret key, and the header block of each application has a symmetric session key. The XOM processor uses its secret key to decrypt the session key, and the processor contains a session key table that has one entry for each process. All data is tagged with an identifier that acts as an index into the session key table. Only the active process can read from or write to memory, and a process can only access memory if the identifier of the process matches the tag. This prevents a process from accessing the memory of another process.

Another proposal to prevent tampering and piracy is to embed encrypted data into each block of instructions prior to installation [Kirovski 02]. In this scheme, the processor checks the integrity of data at runtime using its asymmetric key. Since millions of instructions execute every second, software decryption can become expensive. This has led to proposals to remove software decryption from the critical path so that it can be performed in parallel with other tasks. For example, Yang, Zhang, and Gao propose using AES to generate a one-time pad for software decryption [Yang 03].

The main problem with these three proposals is that if the asymmetric secret key of one processor is discovered, it is possible to foil this scheme by decrypting the software for everyone. This has led to the proposal of different security schemes such as broadcast encryption, which is an anti-piracy system used in DVD players that is based on symmetric crypto [Lotspeich 02]. A DVD player needs the management key in order to play the DVD. The DVD header contains a matrix containing this key encrypted repeatedly with many different device keys, and every device has a unique set of several device keys. If a traitor is discovered, that key is crossed out, and the device tries a different key. One drawback of this scheme is that it requires a central authority.

As we have seen, protecting keys from the adversary is crucial. Virtual Secure Coprocessing is a clever proposal to solve this problem. A virtual secure processor is a general-purpose processor with a special mode for secure execution [McGregor 04]. Only the trusted software library can access the user’s key ring. When trusted code enters the on-chip cache, it is verified using a hash engine. The processor will not execute code unless the verified bit is set. Sensitive data is encrypted prior to leaving the chip using an encryption engine, and only secure mode threads can access a cache line if the sensitive bit is set.

II. C. Possible Applications of Cryptographic Primitives

II. C. i. Applying Crypto to CA

Many cryptographic processors have been invented. These are examples of using architecture to help cryptographic applications run faster. On the other hand, using cryptography to design better architectures is much less common [Smolens 04] [Suh 03]. Computer architects often rely on heuristics that only work well probabilistically. For example, counting bloom filters have the problem of false positives, and some branch prediction schemes rely on machine learning techniques. It would be beneficial to have guarantees on the accuracy of these heuristics in the worst case. Cryptographic primitives such as cryptographic hash functions and incremental hashing [Bellare 96] can provide such guarantees, and they are lightweight enough to demand low overhead.

II. C. ii. Cryptographic Hash Functions

Cryptographic hash functions transform a relatively large string into a much smaller string of constant size. Unlike traditional hash functions, however, cryptographic hash functions guarantee that the probability that two different strings will have the same hash value is infinitesimal. A malicious person must not be able to generate an alternative string with the same hash value in a reasonable amount of time. Hash functions are useful in constrained environments such as embedded systems because a hash value does not take up much space. Unfortunately, any time a large file is hashed repeatedly, the entire file must be read from disk even if only a small part of the file has changed. Incremental hashing provides a way around this problem.

II. C. iii. Incremental Hashing
Incremental hashing is a secure way of generating a new hash value of a string when only a part of the string has been modified. The string is divided into chunks, and the incremental hashing algorithm can generate a new hash value from the old hash, the chunk prior to modification, and the chunk after modification. Chunks that have not been modified do not need to be fetched from storage.

One implementation of incremental hashing involves using the MD5 cryptographic hash algorithm [Rivest 92]. To hash a string, the string is first divided into chunks, and each chunk of the string is assigned a unique identifier. Each chunk and its ID are concatenated and then hashed using MD5. All of the hashes are combined together using addition (modulo M) to produce the hash value H for the entire string. If a particular chunk is modified, the hash of the original chunk (and its ID) is subtracted (modulo M) from H. Then, the hash of the modified chunk (and its ID) is added (modulo M) to produce the hash value H' for the modified string.

![Figure 1: Incremental hashing can update the hash value of a buffer that has been modified slightly without having to re-read the entire buffer. The MD5 hash of the original chunk is subtracted from the original hash value, and then the MD5 hash of the modified chunk is added to yield the new hash value.](image)

II. C. iv. Prior Art Applying Cryptographic Hash Functions to CA

Incremental hashing has been proposed for memory integrity verification [Suh 03]. Assuming that the processor is trusted and that the memory is not trusted, this scheme keeps two hash values: ReadHash and WriteHash. The scheme updates ReadHash on a cache miss, and it updates WriteHash on a cache eviction. It is necessary to “stop the world” in order to check the integrity of memory because everything that is not in the cache must be fetched from RAM and incorporated into the appropriate hash value. If integrity has not been violated, ReadHash should equal WriteHash at the end of this process.

Fingerprinting is another proposal for applying cryptographic hash functions to CA [Smolens 04]. In this scheme, the computation of a pair of processors is cross-checked by representing the architectural state of each processor as a hash value. If the hash values differ, then an error has occurred in one of the processors.

II. C. v. Copy-on-Write Schemes

We believe that incremental hashing can also be used to reduce the number of redundant copies of data required in schemes that rely on “copy-on-write.” For example, synchronizing the data between two computer hard drives can require a lot of bandwidth [Sapuntzakis 02]. Suppose two hard drives start out in the synchronized state, and a program writes to several pages on one of the drives. Under the copy-on-write policy, these pages must be transferred whether or not they differ from the version on the other drive. On the other hand, if we use incremental hashing, we can compare the hashes of the pages that underwent a write, and it is only necessary to transfer those pages with differing hash values. Incremental hashing is especially helpful in reducing the overhead of this scheme on disk performance because only the modifications to the pages need to be hashed. An optimal page size can be chosen that is large enough to take advantage of the benefits of incremental hashing and small enough to fit in the cache.

Besides synchronization, other copy-on-write schemes that may also benefit from incremental hashing include speculative multithreading [Zilles 02]. In speculative multithreading, for example, threads fork speculatively and then roll back if a conflict occurs. A speculatively forked thread that modifies data will cause the system to create another copy in case rollback occurs. Using incremental hashing can reduce the number of redundant copies of data by deleting those copies that have the same hash value as an existing copy.

II. C. vi. Diva

We have already seen cryptographic hash functions used to compare the architectural state of two processors [Smolens 04]. We believe that
it is also possible to apply incremental hashing to the problem of processor reliability. The Diva architecture employs a small checker unit to verify the correctness of the computation of an out-of-order processor [Austin 99]. Verifying the correctness of the checker unit is sufficient to ensure the accuracy of the computation. It is much less expensive to verify the checker unit than to verify the entire processor. Cryptographic primitives could be used to perform a similar function as the checker unit. With incremental hashing, the order of events does not affect the value of the hash. This feature could be exploited to represent the final state of an out-of-order processor’s computation with a hash value, which will be the same regardless of the order in which instructions execute.

III. Application of Incremental Hashing to Memory Allocation Analysis

III. A. Leak Detection

One application of incremental hashing that we have evaluated is leak detection. We use binary instrumentation to intercept every call to malloc(), calloc(), realloc(), and free(). When an allocation occurs, the address A\textsubscript{1} of the newly allocated memory is hashed using MD5 to produce H\textsubscript{1}. If another allocation occurs, this address A\textsubscript{2} is also hashed and added (modulo M) to H\textsubscript{1} to produce H\textsubscript{2}. If a de-allocation of address A\textsubscript{2} occurs next, the hash of A\textsubscript{2} is subtracted (modulo M) from H\textsubscript{2} to produce H\textsubscript{3}. At this point H\textsubscript{3} should equal H\textsubscript{1}. If a de-allocation of A\textsubscript{1} occurs next, the hash of A\textsubscript{1} is subtracted (modulo M) from H\textsubscript{3} to produce H\textsubscript{4}, which should equal zero. This indicates that no memory has leaked so far.

Unfortunately, there is a problem with using incremental hashing for leak detection. Suppose we have five allocations of A\textsubscript{1}, A\textsubscript{2}, A\textsubscript{3}, A\textsubscript{4}, and A\textsubscript{5} followed by two de-allocations of A\textsubscript{1} and A\textsubscript{2}. In this case, A\textsubscript{3}, A\textsubscript{4}, and A\textsubscript{5} were never de-allocated, which means that a memory leak has occurred. At this point in the program the hash value we have is H\textsubscript{7}, which is a combination of the hashes of A\textsubscript{1}, A\textsubscript{2}, and A\textsubscript{3}. There is no way to determine which addresses contributed to H\textsubscript{7} other than trying all possible combinations of one address C(5,1)=5, all possible combinations of two addresses C(5,2)=10, all possible combinations of three addresses C(5,3)=10, all possible combinations of four addresses C(5,4)=5, and all possible combinations of five addresses C(5,5)=1, for a grand total of 32 combinations.

III. B. Allocation States

A better use of incremental hashing is to consider the intermediate hash values H\textsubscript{1}, H\textsubscript{2}, H\textsubscript{3},...,H\textsubscript{i},...,H\textsubscript{N} as discrete states of memory allocation. We can represent these states as nodes in a graph. Allocations and de-allocations are the transitions between the states, and they are represented as directed edges. A solid edge represents an allocation, and a dashed edge represents a de-allocation. If a cycle occurs in this graph, then no memory leaked within that cycle. A well-written program with no memory leaks will always return to the initial state, but we have found that very few programs actually behave this way. A program with very few cycles is probably leaking memory, while a program with many cycles is probably not.

III. C. Prior Art in Memory Graphs

Zeller uses a unique style of memory graph to find bugs in computer programs [Zeller 02]. In this scheme, nodes in the memory graph represent values or variables, and edges represent struct member access, array element access, variable access, and pointer dereferencing. Bugs are found by comparing the memory graph for a working iteration of a program against a version of the program that fails.
III. D. Finding More Reoccurring States

It is possible to search for additional reoccurring states. One way of doing this is to subtract (modulo M) from every $H_i$ the hash of every address $A_1,A_2,A_3,\ldots,A_j,\ldots,A_N$ to compute differences $D_1,D_2,D_3,\ldots,D_j,\ldots,D_N$. If there is a match between some $(D_j,H_i)$ pair, we draw a solid green edge from $H_j$ to $H_i$. We call this search “off-by-one.” The graphs in Figures 4 and 5 are from identical sections of the blackjack program, but the graph in Figure 6 has additional edges in green found by this search. A blue edge indicates a new edge that is also along the shortest path. Since most new edges are usually between nearby states, we can speed up the search by only considering states within a radius of 50. We call this search “limited off-by-one.”

III. E. Shortcut Index

One way of measuring the number of reoccurring states in a program is to use Dijkstra’s algorithm to compute the shortest path from the initial state to the final state. We indicate the shortest path in red. When computing the cost of a path, allocations have a weight of 1, and de-allocations have a weight of 0. Since Dijkstra’s algorithm will bypass any cycles, a program with many reoccurring states will have a small min cost path relative to the total number of allocations in the program. Our metric for measuring the number of reoccurring states is called the shortcut index, and it is computed by taking the ratio of the Dijkstra min cost path divided by the total number of allocations and then subtracting this ratio from 1.
Figure 8: The shortcut index measures the number of reoccurring states for a particular benchmark. Additional states can be found during a search.

IV. Conclusions and Future Work

Applying CA to crypto and vice-versa is a fertile research area. For example, reconfigurable hardware is vulnerable to attacks in which a malicious design is uploaded to the FPGA, causing it to short-circuit. Asymmetric crypto could be used to prevent unauthorized entities from modifying a reconfigurable device. Another issue that merits further investigation is resource sharing by different modules on an FPGA. There needs to be a way to restrict which region of memory a module can access so that one module does not have access to the memory space of another module.

V. References


