DABLS: Device Attestation with Bounded Leakage of Secrets

Andrew Tran

July 12, 2013

CMU-CyLab-13-010

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

DABLS: DEVICE ATTESTATION WITH BOUNDED LEAKAGE OF SECRETS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF CARNEGIE MELLON UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTERS

Andrew Tran

July 2011

Contents

Abstract

Use of commodity platforms for embedded systems makes it difficult to authenticate remote devices in the presence of malware and to obtain confirmation of malware-free device states in a verifiable manner. We propose a scheme for achieving these properties by installing and maintaining a pool of secrets in device memory that cannot be leaked by malware in its entirety via a bandwidth-limited (e.g., wireless) channel during a specified time epoch. Correct device operation limits malware leakage of pool content by updating the pool with fresh secrets. It is computationally infeasible for the adversary to compute the new pool given the limited information he was able to leak about the old pool within the specified time epoch. Verifier detection of a device's failure to update the pool in a timely manner indicates the presence of active device malware and triggers remedial action (e.g., automated pool-content update, or manual device cleanup). Verified timely pool updates provide device authenticity, since all devices are initialized with independent pool secrets (i.e., pseudorandom values), and enable bringing the remote device to a malware-free state by removing malware from device memory. In this paper, we elaborate on these ideas and illustrate how our system complements the goals of cryptographic schemes that are resilient to continual but bounded secret-key leakage via side channels.

Chapter 1

Introduction

Large-scale embedded systems (e.g., power distribution and industrial control applications) make pervasive use of commodity platforms and devices, wireless communication infrastructures, and remote system management – all for strong economic reasons. For example, remote management avoids the need for frequent local embedded-device access during normal operation, while wireless network connectivity lowers the cost of device deployment in scalable configurations. Use of commodity platforms reduces development costs to those of system integration and drastically lowers maintenance costs. However, pervasive use of commodity platforms and devices, wireless communication infrastructures, and remote device management pose significant security challenges.

Cost pressure on embedded systems frequently leads to the use of low-cost devices, which lack specialized security hardware (e.g., TPM, random number generator support), and low-assurance software, which often leaves the system vulnerable to untraceable malware attacks. Furthermore, hardware-based cryptographic support for device security requires sophisticated key management and revocation strategies, which are expensive to deploy and maintain, and highly prone to human error. However, remote device management requires "over-the-air" device authentication before administrative commands can be executed in a secure manner. To date, neither remote authentication of *low-cost* commodity devices in the presence of malware nor *remote* o attestation of a device's

malware-free state has been achieved in practice.[1] Achieving both of these properties on commodity platforms and low-cost devices without specialized security hardware in a demonstrable manner is the goal of this paper.

We consider a network setting where a trusted verifier communicates with a remote, commodity device via a bandwidth-limited (e.g., wireless) channel. More concretely, the verifier could be a utility company and the device could be a "smart meter" deployed in the field [9]. The verifier can be trusted since it is under close administrative control and can leverage specialized secure hardware on a central server [12, 29]. In contrast, the remote device may only be assumed to be in a malware-free state at the time of deployment. Past that point, the device may become infested with malware. Nevertheless, at a minimum, the verifier wants to obtain (1) assurance of device authenticity despite the presence of malware, and (2) attestation of malware-free device state at various times during device operation; e.g., during remote software updates and device reboot. With these properties, we envision an architecture that prevents malware-based device cloning and impersonation, enables secure management of remote devices, and achieves verifiable application secrecy and integrity, despite the absence of any specialized hardware or long-term secrets.

However, achieving the two properties mentioned above is challenging. For example, establishing remote device authenticity requires that device secrets be protected from exfiltration by malware and potential reuse on foreign devices – a tall order on commodity platforms that lack specialized hardware protection. Remote attestation of malware-free device state requires demonstrable removal of potentially active malware from a device's memory – a challenging task even via prior approaches such as local software-based attestation and device reset [6, 21, 25].

The basic idea that underlies our approach to remote device authentication and attestation of malware-free state is to install and maintain a pool of secrets in device memory, that cannot be leaked by malware in its entirety via a limited-bandwidth (e.g., wireless) channel during a specified time epoch. Thus, the only hardware requirement for our scheme is the availability of sufficient

[1] These properties can be achieved only on devices equipped with special security hardware, often requiring considerable engineering effort to integrate with existing software and systems [12, 29].

device memory – our security mechanisms are entirely software-based. In addition, attestation of malware-free device state requires a small read-only-memory to store the protocol code itself. Once per epoch, the device updates the secret pool and obtains a fresh pool. Epochs are sized such that it is computationally infeasible for an external adversary to compute the fresh pool given the limitations on what information he was able to obtain about the previous pool. The remote device attests to having updated its pool in a timely fashion. For the pool updates, we use a non-invertible, pseudo-random function that has strong, non-circumventable, time-space tradeoffs. Thus, malware cannot enlarge memory leakage and still perform timely pool updates. If the verifier detects a device failing to update the pool within a time epoch, this indicates the presence of active malware on the device and triggers remedial action (e.g., a remote pool reset attempt, or manual device cleanup).

Verified timely pool updates provide device authenticity, since all devices are initialized with independent secrets (i.e., pseudo-random values), and can enable bringing the remote device to a malware-free state by removal of certain classes of malware from device memory. Note that malware with total control over the device can refuse to participate in our protocols, resulting in a denial of service attack on the remote device. However, this is readily detected by the verifier. Malware wishing to remain stealthy will be forced to relinquish its position on the device.

The idea of bounding continual leakage of secrets is reminiscent of a somewhat similar notion used in constructing leakage-resilient cryptographic schemes [11, 15, 22, 14]; viz., Section 2.2 (Related Work). That is, secret keys corresponding to the same public key can change in well defined time epochs before they become vulnerable to side-channel attacks. However, all these schemes assume that the key update process can leak only a limited number of key bits and that the adversary (i.e., malware, in our case) cannot interfere with that process; e.g., cannot corrupt the source of local randomness or leak local random values; cannot hide old keys in obscure memory locations for the purpose of future leakage and circumvent the old-key-deletion requirement. Exclusive use of these schemes cannot be made to achieve our desired properties. However, achieving our properties can, in fact, support the assumptions made by these schemes and enable them to achieve their goal (i.e., protection from side-channel attacks) in the presence of malware on low-cost devices and

commodity platforms.

In short, the main contributions of this work as as follows: (1) we verify remote-device authenticity despite the presence of malware (e.g., detect attempted device cloning and impersonation), and provide for the remote attestation that device state is malware-free; (2) we support secure updates of remote devices offering code and data integrity; and (3) we enable secure invocation of commands on remote devices. All these desirable system feature are provided on low-cost devices that lack specialized secure hardware.

Chapter 2

Background

We provide background on related work. We will briefly describe how the previous schemes work and the differences between our work and the previous approaches.

2.1 Attestation

We provide background on code attestation and related cryptographic work. Code attestation is a technique for verifying what code runs on a system. There are both hardware and software methods for code attestation.

2.1.1 Software attestation

We examine previous software attestation schemes that use challenge response protocols to verify the memory of a external device. The challenge of designing such protocols is that the adversary has control of the device during the verification computation.

SWATT [26] is a method that uses tight timing constraints to identify malware. SWATT traverses memory pseudo-randomly by using a seed sent by the verifier. In order for the malware to pass verification, it needs to redirect memory accesses away from memory regions where malware resides. The insight is that the check to perform a memory redirection will happen every single

memory access regardless if the memory is actually redirected. SWATT makes enough pseudo-random memory accesses such that the malware's check will induce a externally measurable timing gap between malware-free code and malware infected code. ICE [23] extends the concept by using additional CPU state to provide stronger properties. Also, ICE only needs to verify the checksum versus verifying all of memory. The difference between our scheme and SWATT/ICE is that our scheme also enables secrecy to be a verified in addition to code integrity. Our scheme also does not rely on tight timing measurements to distinguish between malware and benign code.

Secure Code Update for Embedded Devices via Proofs of Secure Erasure [21] is a method to locally attest to malware free state. This software attestation scheme attests to a malware free state locally by filling all writable memory with pseudo-random values. Their scheme also not rely on tight timing constraints. Their scheme does not allow the prover to communicate to other devices during verification. Our scheme has lower bandwidth costs because during verification our verifier only sends small nonce versus sending enough pseudo-random values to fill all writable memory. Our scheme also considers the harder problem of verifying malware free states remotely.

Remote Software-Based Attestation for Wireless Sensors [27] uses a different software attestation routine every time. The verifier sends a different attestation routine for each attestation. Each attestation routine the verifier sends is protected by code obfuscation and self modification to prevent modification by the attacker. The protection techniques combined with limited time to perform the attestation make it hard for the attacker to reverse engineer the routine and forge a correct response to the attestation. The security of this approach relies on the hardness of code obfuscation which may be hard to prove secure. Our approach relies on properties which may be easier to prove secure.

2.1.2 Hardware Attestation

Hardware support for trustworthy computing (i.e., the Trusted Platform Module (TPM) [28] and ARM TrustZone [3]) is currently the preferred approach protecting device secrets on commodity

platforms. While these secrets can be used in remote device authentication and attest to malware-free device states, they can be leaked via side-channel attacks [18, 17]. Hence, the resilient cryptographic approaches described above become necessary. (However, additional mechanisms may become necessary to remove the unrealistic assumptions made by these approaches; e.g., regarding the bounded leakage of secret key material during key updates.) Hardware support for physical authentication has also been provided by Physically Unclonable Functions (PUFs) [16, 20]. While PUFs are very basic authentication primitives, they are not designed to offer malware-free device states.

2.2 Limited Leakage Cryptography

Cryptographic constructs that allow limited leakage of secret material (e.g., secret keys) have been extensively explored recently. Most leakage-resilient cryptographic schemes follow three distinct approaches, namely (1) the bounded retrieval model, (2) the leakage-resilient cryptography, and (3) the continual-leakage-resistant cryptography. In this section, we point out the specific differences between our work and these approaches. To provide further perspective, we also discuss related work in the area of trustworthy computing in general, and software-based attestation in particular.

2.2.1 Bounded Retrieval Model

The bounded retrieval model [8, 13, 5, 1, 2] assumes that an adversary can recover some function f of a large secret S. The adversary may choose to leak S over multiple rounds either non-adaptively, using the same function f per round, or adaptively, by picking different a different f in each round. This model restricts the adversary leakage over all rounds to $|S|$. Our adversary model is more general in the sense that the total amount of leakage may exceed $|S|$; i.e., malware may leak $r \cdot L > |S|$ bits, adaptively or not, where r is the number of rounds and L is the maximum number of bits leaked per round (viz., Figure 1).

2.2.2 Leakage-Resilient Cryptography

Leakage-resilient cryptography [11, 15, 22, 14] assumes that an adversary can adaptively recover a secret S over multiple rounds by picking different functions f in each round. The adversary is restricted in two ways, namely (1) in every round r, no more than L bits of the secret S_r may be leaked, and (2) the bits of S_r can be leaked only by the computation at round r, and cannot be memory bits. (This model makes the "only computation leaks information" assumption originally proposed by Micali and Reyzin [19]). Our adversary model is more general, since it does not restrict which secret bits malware may leak in a given round. As mentioned above, we only assume that no more than L bits of the secret S_r may leak in any round r.

2.2.3 Continual-Leakage Resistant Cryptography

Continual leakage resistant cryptography [10, 4] also assumes that an adversary can may adaptively leak secret S over multiple rounds by picking different functions f per round. Similar to our approach, this model updates the secret in each round r to ensure that no more than L bits of the secret S_r may leak. However, this model is less general than ours since it assumes that the adversary is only able to probe the device but does not have full control over it; e.g., over the update function for S_r. That is, the update function (1) uses local randomness unavailable to the adversary and leaks only a limited number of bits during its operation, and 2) is protected from adversary interference; e.g., the adversary may not corrupt or stop the update of secret S_r by modifying the local, per-round random bits used, nor it allowed to copy, save and reuse old secrets (i.e., old secret keys). In contrast, we let our adversary have *full control* over the device software, including the update function and secret pool S. If the adversary updates the secret pool in a corrupt manner (e.g., incompletely, or not at all) or the update exceeds the allowed time limit, the device cannot respond (correctly) to the verifier's challenge and is reset.

2.2.4 Trustworthy Computing Hardware and Software-Based Attestation

Hardware support for trustworthy computing (i.e., the Trusted Platform Module (TPM) [28] and ARM TrustZone [3]) is currently the preferred approach protecting device secrets on commodity platforms. While these secrets can be used in remote device authentication and attest to malware-free device states, they can be leaked via side-channel attacks [18, 17]. Hence, the resilient cryptographic approaches described above become necessary. (However, additional mechanisms may become necessary to remove the unrealistic assumptions made by these approaches; e.g., regarding the bounded leakage of secret key material during key updates.) Hardware support for physical authentication has also been provided by Physically Unclonable Functions (PUFs) [16, 20]. While PUFs are very basic authentication primitives, they are not designed to offer malware-free device states.

Software-based attestation has been proposed to achieve code and execution integrity on commodity platforms without special hardware support [26, 24, 30, 7, 27]. These approaches can leverage our mechanisms to protect remote attestation against proxy-attacks. Moreover, we greatly relax their timing and hardware architecture assumptions, which have been exploited by recent attacks [6]. Our approach is immune to these attacks since we do not rely on micro-architectural details of the hardware platform (e.g., timing of instruction execution).

Another notion related to attestation of malware-free device states was proposed by Perito and Tsudik [21]. Their approach enables a safe local device reset by filling the device memory with a pseudo-random pattern. Subsequent local verification of the pattern ensures secure memory erasure. Secure *remote* operation is not a relevant goal of this work. In contrast, our approach is intended to address the more challenging case of secure *remote* device authentication and reset to malware-free states.

Chapter 3

Design

3.1 Assumptions and Attacker Model

3.1.1 Assumptions

We make hardware assumptions about the embedded device our system runs on.

Physical Security Assumption. We assume that the device is physically secure, i.e., it is in a safe location, or else it is constructed using sufficient physical anti-tamper mechanisms. Such mechanisms are orthogonal to our scheme and outside the scope of this paper.

Fixed Bandwidth Assumption. We assume that the target device D has a fixed maximum *uplink* bandwidth D_{ban} on its network connectivity with the outside world. It is physically infeasible for an adversary to exfiltrate data at a greater rate than the device's maximum bandwidth. We assume that the adversary cannot alter the hardware (e.g., by adding additional interfaces or replacing an existing interface with one capable of higher bandwidth).

Complete Channel Assumption. We assume the the network interface referenced in the Fixed Bandwidth Assumption is the device's only network interface. If the device has additional interfaces, then without loss of generality we can consider them in aggregate, and apply the Fixed Bandwidth Assumption to the aggregate network interfaces. I.e., D_{ban} is unchanged.

Fixed Memory Assumption. We assume that the target device D is deployed with a fixed amount of memory D_{mem}, and that the adversary is incapable of adding additional memory to the device. I.e., the maximum amount of memory in the device is fixed.

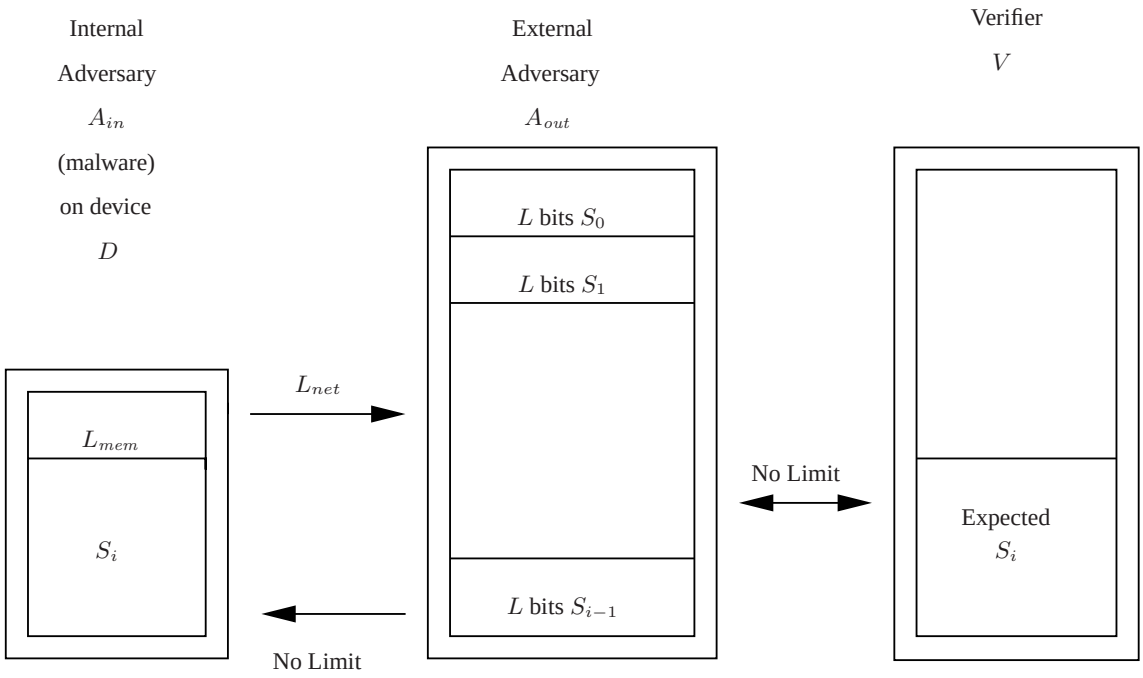

Figure 3.1: DABLS state during epoch i.

3.1.2 Attacker Model

Our attacker model includes both an internal adversary, A_{in}, and an external adversary, A_{out}. Both are probabilistic, polynomially-bounded adversaries and, as such, cannot break any of the standard cryptographic primitives that is proved secure for such adversaries. A_{in} has compromised the target device D, and is capable of running arbitrary code (i.e., malware) on D. A_{in} can communicate with external adversary A_{out} via device D's network interface and can cooperate with A_{out} at all times. The resources available to A_{in} are the device D's maximum outbound network bandwidth, D_{ban}[1], and maximum memory D_{mem}. In particular, A_{in} has unrestricted access to the code and

[1]We do *not* constrain the inbound bandwidth to D.

workspace (e.g., stack) of our remote-device authentication protocol in D_{mem}, and thus can leak and modify their content. Furthermore, A_{in} has unrestricted access to all D_{mem} workspace of our protocol for achieving malware-free device states, but not to its instructions, which execute in a device read-only memory (ROM). By the physical-access assumptions made above, the adversary cannot modify device D's hardware to increase D_{ban} or D_{mem}, and thus it must develop an attack strategy that makes efficient use of these resources.

To illustrate the adversary's operation, we break time into epochs of duration T. Thus, the maximum amount of data that can be exfiltrated by A_{in} to A_{out} during an epoch, which we call the *network leakage*, is $L_{net} = T \cdot D_{ban}$. Additionally, we divide D_{mem} into two areas: one that stores a large pool of secrets that is refreshed during every epoch i, which we denote by S_i, and the other, which we call the *memory leakage*, $L_{mem} = D_{mem} - |S_i|$. The adversary can make full use of L_{mem}, and hence it can use it to save pool data to be leaked during one or more future epochs.A_{in} can also perform arbitrary (polynomially bounded) computations on any S_i and can communicate the results of those computations to A_{out}. Over time, A_{out} may accumulate data that is leaked from D, at a maximum rate of L_{net} bits per epoch subject. Thus, the maximum amount of data that A_{in} can ever leak from a given epoch's pool to A_{out} is $L \leq L_{net} + L_{mem}$. Figure 3.1 summarizes the state of the system at epoch i.

3.1.3 Taxonomy of Attacks

We explain several attacks A_{in} and A_{out} can perform on our scheme. The goal for all of the attacks is for A_{out} to learn more than L bits of any secret pool S_i used by the device D. The attack will explain are the rolling pool attack, the split computation attack, the split computation attack and the time space trade-off attack.

Rolling Pool Attack

We give a brief overview of the rolling pool attack. The rolling poll attack is where the internal adversary A_{in} rolls a secret pool S_i to past or future pools to leak those pool states during epoch

i. The internal adversary A_{in} by leaking past or future pools can leak more than L bits of a past or future pool to the external adversary A_{out}, therefore external adversary A_{out} will be able to learn more than L bits of a past or future pool.

Split Computation Attack

We give a brief overview of the split computation attack. At epoch i, the internal adversary A_{in} leaks l blocks of pool i to the external adversary A_{out}. Then during the pool update from pool i to pool $i+1$ the internal adversary A_{in} leaks special information to the external adversary A_{out}. The special information is defined by anything the internal adversary A_{in} knows that would allow the the external adversary A_{out} to update the leaked blocks of pool i to pool $i+1$. The special information could be intermediate values used in the update computation. Finally, external adversary uses the special information that A_{in} leaked to update its pool i blocks. The external adversary A_{out} By updating blocks of pool i in its possession, the external adversary A_{out} could learn more than L blocks of a secret pool i. The adversary's advantage in performing the split computation attack against the update from i to $i+1$ is the number of blocks the external adversary A_{out} was able to update from pool i to $i+1$.

Time Space Trade-off Attack

We give a brief overview of the time-space trade-off attack. In the time space trade-off attack, the internal adversary A_{in} re-implements the pool update function f to store less than $|S|$ blocks in memory for both computation and output. One might think the hash verification would be able to detect if the output of f occupied less than $|S|$ blocks of memory. However, internal adversary A_{in} can compute the inputs to the hash verification *on demand* without simultaneously storing $|S|$ blocks in memory. The advantage A_{in} gains by performing this attack is the ability to store more blocks of a pool to leak during later epochs.

3.2 An Unleakable Secret Pool

In this section we first provide an overview of DABLS, and then provide a more detailed and formal treatment.

3.2.1 DABLS Overview

In DABLS, a remote device is associated with a pool of secrets, which defines the identity of that device. The pool can be used as the key to a message authentication code computation to authenticate the device to the remorte verifier. Hence, a basic goal of DABLS is to prevent an external adversary A_{out} from learning enough of a device's secret pool from an internal adversary (malware) A_{in} so that it could discover the entire pool in a computationally feasible time. In other words, we say that a secret pool is unleakable during the lifetime of the device, if the maximum pool leakage from any epoch i, namely L, is sufficiently smaller than the size of the pool, $|S_i|$, so that A_{out} would find it computationally infeasible to recover the entire secret S_i. We denote this relationship between L and S_i as $L << S_i$.

We achieve this property through periodic protocol interactions by a remote verifier, which require that the device attest that it updates its secret pool in a verifiably correct and timely manner. This ensures that the secret pool changes its secret pseudo-random contents faster than A_{in} could leak more than L bits of pool S_i for all epochs i of the device's lifetime. We parameterize the protocol invocation and verification frequency based on the pool size $|S|$ and the device's available bandwidth D_{ban}, so that we can assure that a maximum of L bits can possibly be leaked even if malware is present on the device.

Pool Size Requirements A secret pool S_i must be large with respect to the device's network bandwidth D_{ban} over a defined time epoch, T. Even with this requirement, a patient adversary A_{in} may eventually leak the entire secret pool. This gives rise to the dual requirements that (1) the pool's contents must change over time to comprise pseudo-random values, and (2) the amount of

the device's memory that the pool occupies must be specified as a function of L_{net} and L_{mem}. That is, A_{in} must not be able to exfiltrate any single epoch's complete pool via L_{mem} and L_{net}. Thus, in a particular epoch i, we require that $L_{net} + L_{mem} < |S_i|$. Substituing the definition of L_{mem}, we have $L_{net} + (D_{mem} - |S|) < |S|$, which simplifies to:

$$|S_i| > (L_{net} + D_{mem})/2$$

If A_{out} receives precisely $(L_{net} + D_{mem})/2$ bits of S_i from A_{in}, it will still find it to be computationally infeasible to recover the remainder of S_i.

Pool Update Requirements. To maintain the unleakable properties of pool S_i beyond a single epoch, any of its bits that A_{out} may have accumulated during an epoch i should become useless to A_{out} following the pool update by a *pool update function* f. To achieve this, we invoke f periodically, in response to a verifier's fresh request. We divide time into *epochs* of duration T, and require that the pool update from S_i to S_{i+1} once per epoch. A delinquent pool update constitutes a misbehaving device. The inital pool, S_0, is injected by a trusted party under controlled conditions before malware has the opportunity to infect the device.

Protocol Requirements The verifier challenges the device to update its secret pool correctly and timely within an epoch, and verifies the response received from the device. As our attacker model gives A_{out} complete access to the network, A_{out} can oberve any messages used in the challenge-response and pool-update protocols. Thus, if the external adversary A_{out} would eventually be able to construct *any* complete instance of a pool S_i, then A_{out} can use its log of messages and compute the pool at a later epoch $k > i$.

We seek to ensure that any L pool bits or fewer of an older epoch i that an adversary has captured are of no value to the adversary in trying to compute the current-epoch pool S_k, where $k > i$. Thus, the output of the pool update function f must be indistinguishable from random to the (polynomially bounded) adversary, and be of the same size as the existing secret pool, i.e., $|S_i| =$

$|S_{i+1}|$. Otherwise, the adversary may find ways to compress the representation of S_i, effectively increasing D_{ban}.

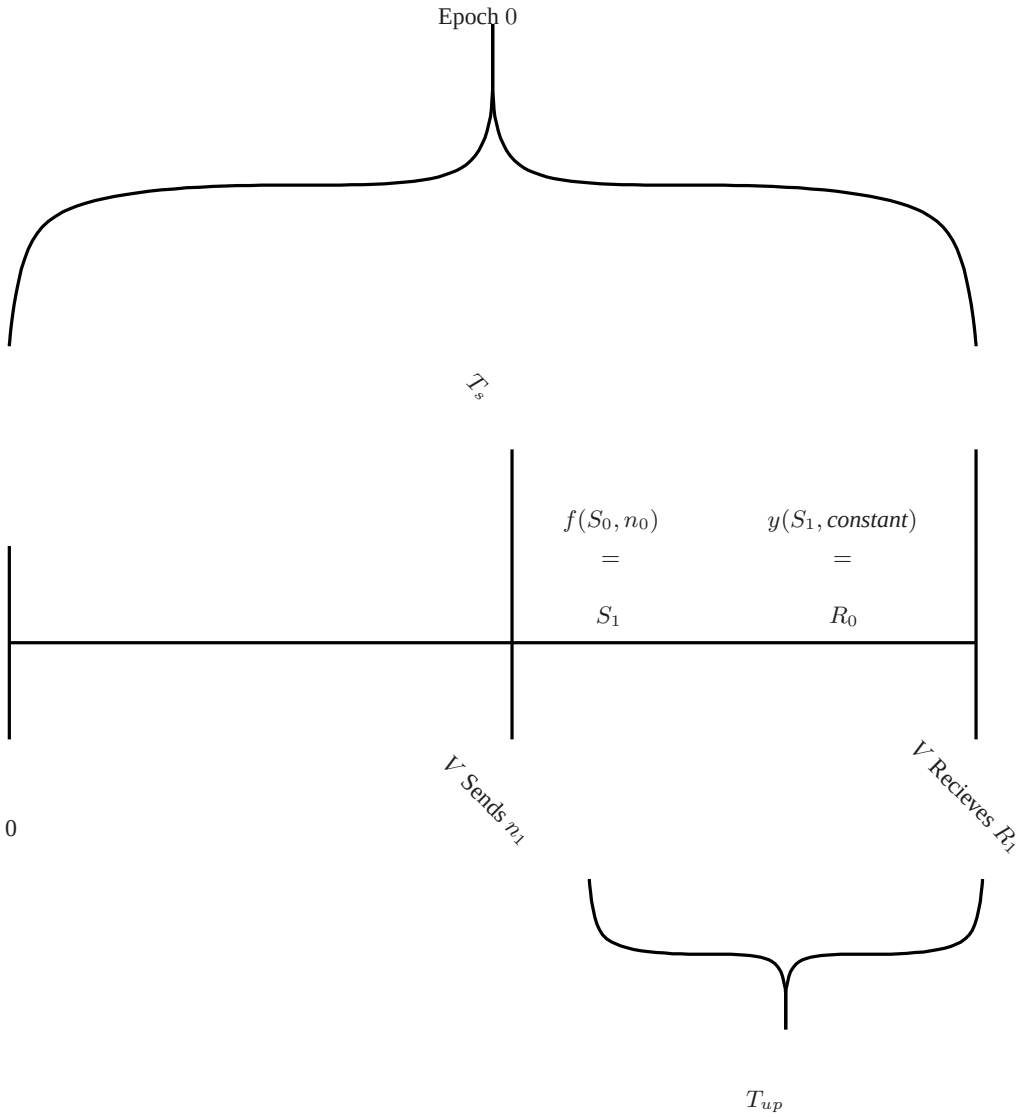

Figure 3.2: Initailzation at Epoch 0 Timeline

Timing Requirements An unleakable secret only lives for a single time epoch of duration T. We define the time at the start of the i^{th} epoch as iT_s. By the end of the epoch, the pool must

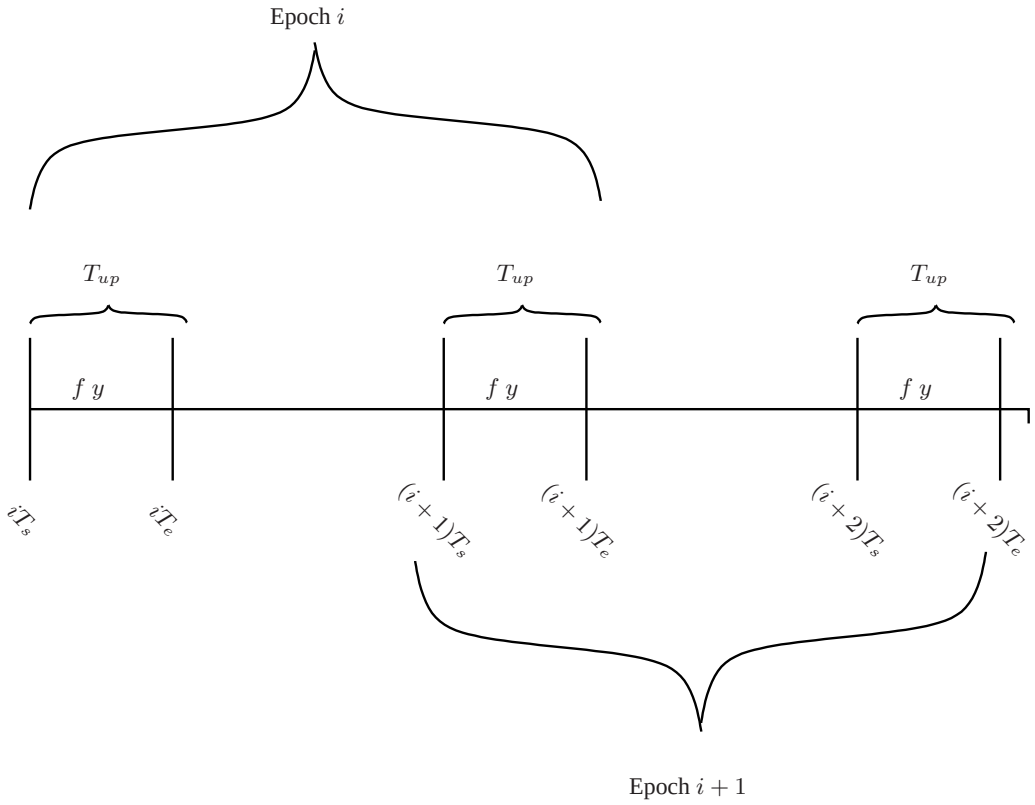

Figure 3.3: Steady State at all subsequent Epochs i.

be updated. D has a limited amount of time T_{up} to compute the pool update to prevent A_{in} from leaking arbitrarily large amounts of data. The verifier enforces the time limits by releasing the nonce n_i only at time $(i+1)T_s$.

Figure 3.2 the shows a timeline of the update process from epoch 0 to epoch 1. Epoch 0 starts at time 0. After T_s time, the verifier sends the nonce n_i to D starting the update process from epoch 0 to 1. D now computes $S_1 = f(S_0, n_0)$. Then D computes $R_0 = y(S_1, \textit{constant})$. D sends R_0 to V by time $T_e = T_S + T_{up}$. V checks if R_0 is correct and received by time T_e, if so, then the verifier is confident that D correctly updated S_0 to S_1 in a timely manner thus both epoch 0 and the update from epoch 0 to 1 are finished, otherwise the verifier has detected malware on the system.

Figure 3.3 shows how epochs overlap. Epoch i starts at time iT_s and does not end until the end of the ith update at time $(i+1)T_e$. Epoch $i+1$ starts when the ith update begins i at time

$(i+1)T_s < (i+1)T_e$. Therefore, epochs i and $i+1$ overlap during the ith update process. The epochs i and $i+1$ overlap during the ith update process because during the ith update D has access to portions of both S_i and S_{i+1}.

DABLS Protocol In DABLS verifier V initializes device D with the first unleakable secret S_0 before D is connected to the network and exposed to attack. The pool update during each epoch consists of two functions: f and y. f rolls foward the pool from S_i to S_{i+1}. f takes as input the current pool and a verifier-supplied nonce n_i and outputs the next pool: $S_{i+1} = f(S_i, n_i)$. The primary responsibility of f is to ensure that the pool changes over time. y computes a message authentication code on the output of f that enables the verifier V to ascertain the integrity of the current pool: $R_i = y(S_{i+1}, \textit{constant})$ where *constant* is a block containing a constant string using S_{i+1} as the secret key. To summarize, during each epoch i the verifier V sends challenges n_i and m_i to D (allowing D time to compute f in between), and confirms that the response R_i equals $y(f(S_i, n_i), \textit{constant})$. If R_i is not as expected, the verifier is considered to have detected an attack.

3.2.2 Pool Update Requirements

A_{in} has incentive to cheat by finding ways to compute the update function function f and verification function y (thereby obtaining the correct response R_i) while consuming as few resources as possible. For example, since y is implemented in software and cannot be atomic, A_{in} may search for ways to pipeline the computation of f and y so as to reduce the amount of memory required to produce R_i. This would have the effect of increasing the attacker's L_{mem}, potentially allowing the attacker to gain more bits of S_i. To prevent A_{in} from cheating, f and y need several properties, specified below.

Requirement 3.2.1. ***Non-circumventable Time-Space Tradeoff for f.*** *Any implementation of f concurrently storing less than $|S|$ bits of data not invertible to S_i for computation and output causes the update process to take longer than T_{up}.*

L	Max number of bits that leak about any S
L_{net}	Bits malware can leak in one epoch
L_{mem}	Bits freely usable by malware
$\|S\|$	The set of unleakable secrets
S_i	The unleakable secret at the ith epoch
$\|S\|$	The size of S
D	The device
D_{mem}	The size of the device memory
D_{ban}	The device bandwidth
T	Duration of an epoch
T_s	The time at the start of an epoch
T_e	The time at the end of an epoch
T_{up}	Total time limit to compute the update
f	Pool roll forward function
y	Message Authentication Code Function
R_i	y's output at the ith epoch
n_i	Nonce at epoch i used by f
w	Function over S, n
A_{in}	The internal adversary
A_{out}	The external adversary
V	The Verifier

Figure 3.4: Summary of Notation Used

Requirement 3.2.2. ***f is Non-Invertible****. Given, for all $k < i$, L bits of S_k, nonces n_k, and responses R_k and any S_i, it is computationally infeasible to compute additional bits of any S_k.*

Requirement 3.2.3. ***f Needs Complete Pool****. Given, for all $k < i$, L bits of S_k, nonces n_k, and responses R_k and n_i, it is computationally infeasible to compute any bits of $f(S_i, n_i)$ without all of S_i.*

Requirement 3.2.4. ***f Cannot Compute Early****. Given, for all $k < i$, L bits of S_k, nonces n_k, and responses R_k and S_{i-1}, It is computationally infeasible to compute any bits of $f(S_i, n_i)$ without n_i*

Requirement 3.2.5. ***Leakage Equality for f****. Given, for all $k < i$, L bits of S_k, nonces n_k, and responses R_k and $L - Z$ bits of S_i, Z bits of outputs from any function $w(S_i, n_i)$ computable by A_{in} cannot be used to compute more than Z bits of S_i or S_{i+1} where $S_{i+1} = f(S_i, n_i)$.*

Requirement 3.2.6. ***y is Preimage Resistant****. Given L bits of S, a $constant$, and $R = y(S, constant)$,*

it is computationally infeasible for the attacker to find any additional bits of S.

Requirement 3.2.7. ***y Needs Complete Pool.*** *Given $constant$, for any S_i it is computationally infeasible to compute any bits of $y(S_i, constant)$ without all of S_i.*

3.2.3 Formal Protocol Properties

We prove a statement that limits A_{out}'s knowledge for secrets S_k, $k \leq i$ up to time $(k+1)T_e$, given that A_{out} has received all R_k on by time $(k+1)T_e$.

Theorem 3.2.1. *For all $k \leq i$, if the verifier receives the correct R_k by time $(k+1)T_e$, A_{out} will never know more than $L_{net} + L_{mem} \leq L << |S|$ bits of any S_k.*

To show that A_{out} has limited knowledge of secret pool values S_k for all $k \leq i$, we prove that our protocol limits A_{out}'s knowledge to at most L bits of S_i if V has received the correct and timely response R_k for $k \leq i$.

Theorem 3.2.2. *Given the verifier received the correct R_k by time $(k+1)T_e$ for $k < i$, and if the verifier receives the correct R_i by time $(i+1)T_e$, then A_{out} will never know more than $L_{net} + L_{mem} \leq L << |S|$ bits of S_i.*

The next three lemmas show that our protocol constrains A_{in} to leak at most L bits of S_i before the start of epoch i, during epoch i, and after epoch i.

Lemma 3.2.1. *For time $t < iT_S$, it is computationally infeasible for A_{in} to leak any bits of S_i or $w(S_i, n_i)$ or $w(S_{i-1}, n_{i-1})$, for any function w.*

Proof. The verifier only releases the nonce n_{i-1} at time iT_s. For time $t < iT_s$, A_{in} cannot posses n_{i-1}. Using Requirement 3.2.4, for time $t < iT_s$, A_{in} cannot compute S_i or $w(S_{i-1}, n_{i-1})$ or $w(S_i, n_i)$. Therefore, for time $t < iT_s$, A_{in} cannot leak any bits of S_i or $w(S_{i-1}, n_{i-1})$. □

Lemma 3.2.2. *Given that V receives the correct R_i by time $(i+1)T_e$, for time $iT_s < t < (i+1)T_e$, A_{in} can leak at most L_{net} bits of S_i to A_{out}, where $L_{net} << S_i$.*

Proof. By definition of the DABLS protocol, if V receives the correct R_i by time $(i+1)T_e$, then $(i+1)T_e - iT_s = T$. By the *Fixed Bandwidth Assumption* and *Complete Channel Assumption* the maximal number of bits A_{in} can send to A_{out} during time $iT_s < t < (i+1)T_e$ is $L_{net} = T \cdot D_{ban}$. A_{in} can only leak L_{net} bits of S_i or $w(S_i, n_i)$ or $w(S_{i-1}, n_{i-1})$. By Requirement 3.2.5, the advantages conferred to A_{in} by leaking S_i, $w(S_i, n_i)$, or $w(S_{i-1}, n_{i-1})$ are equivalent. Therefore, given that V receives the correct R_i by time $(i+1)T_e$, for time $iT_s < t < (i+1)T_e$, A_{in} can leak at most L_{net} bits of S_i. □

Lemma 3.2.3. *Given that V receives the correct R_i by time $(i+1)T_e$, for time $t > (i+1)T_e$, it is computationally infeasible for A_{in} to leak more than $L = L_{mem} + L_{net} << S$ bits of S_i.*

Proof. By Requirements 3.2.3 and 3.2.7, R_i can only be computed by the attacker computing $y(f(S_i, n_i), constant)$. If V receives the correct R_i, then f must have computed S_{i+1}. Given that the earliest time when V sends the nonce n_i is $(i+1)T_s$, and that V receives R_i by time $(i+1)T_e$, then A_{in} can take at most T_{up} time to compute the update from S_i to S_{i+1}, because A_{in} could not have computed the update early (by Requirement 3.2.4). Given that A_{in} took at most T_{up} time to compute the update from S_i to S_{i+1}, then we know that the computation of f concurrently stored at least $|S|$ bits for computation and output. Given that computation of f concurrently stored at least $|S|$ bits for computation and output, and that f computed S_{i+1}, then we have that f concurrently stored $|S|$ bits of S_{i+1}.

We know that f concurrently stored $|S|$ bits of S_{i+1}, but we do not know whether A_{in} has chosen to store some of the $|S|$ bits of S_{i+1}. There are only two cases: A_{in} can choose to store the S bits in D_{mem}, or A_{in} can collude with A_{out} to store up to L of the $|S|$ bits on A_{out}.

Suppose A_{in} and A_{out} concurrently store $|S|$ bits of S_{i+1} only in D_{mem}, then we have for time $t > (i+1)T_e$, $|S|$ bits of D_{mem} cannot be used to compute any bits of S_i or $w(S_{i-1}, n_{i-1})$ or $w(S_i, n_i)$ except for $S_{i+1} = f(S_i, n_i)$ by Requirement 3.2.2. For time $t > (i+1)T_e$, given A_{out} only knows L bits of S_{i-1}, A_{in} and A_{out} cannot recover S_i or $w(S_{i-1}, n_{i-1})$ or $w(S_i, n_i)$ except for $S_{i+1} = f(S_i, n_i)$ by Requirements 3.2.2 and 3.2.6. We have established for time $t > (i+1)T_e$, S_i or

$w(S_{i-1}, n_{i-1})$ or $w(S_i, n_i)$ are effectively "deleted" from $|S|$ bits of D_{mem}, and S_i or $w(S_{i-1}, n_{i-1})$ or $w(S_i, n_i)$ can never be recovered by A_{in} and A_{out}.

For time $t > (i+1)T_e$, S_i) or $w(S_{i-1}, n_{i-1})$ or $w(S_i, n_i)$ can still persist in D_{mem} if A_{in} stores a copy of a portion of S_i or $w(S_{i-1}, n_{i-1})$ or $w(S_i, n_i)$ in memory not occupied by S_{i+1}. For time $t > (i+1)T_e$, by the *Fixed Memory Assumption*, the maximum number of bits available to A_{in} to store a portion of S_i or $w(S_{i-1}, n_{i-1})$ or $w(S_i, n_i)$ is $L_{mem} = D_{mem} - |S|$ For time $t > (i+1)T_e$, only L_{mem} bits of A_{in} exist in D_{mem} and S_i or $w(S_{i-1}, n_{i-1})$ or $w(S_i, n_i)$ can never be regenerated, therefore we have for time $t > (i+1)T_e$, A_{in} can never leak more than L_{mem} bits of S_i or $w(S_{i-1}, n_{i-1})$ or $w(S_i, n_i)$ excluding what A_{in} leaked during epoch i. By concurrently storing all $|S|$ bits of S_{i+1} only on D_{mem}, A_{in} did not need to store any bits of S_{i+1} on A_{out}. Therefore, during epoch i, A_{in} could have leaked $L_{net} = T \cdot D_{ban}$ bits of S_i to A_{out}. We have for time $t > (i+1)T_e$, A_{in} cannot leak more than $L = L_{net} + L_{mem} = D_{mem} - |S| + T \cdot D_{ban} < S$ bits of S_i.

The intuition of this case is if A_{in} chooses to store bits S_{i+1} to increase L_{mem} then A_{in} will have to decrease L_{net} because A_{in} can only access A_{out} through the communication channels. Given f concurrently stored $|S|$ bits of S_{i+1} in both D_{mem} and A_{out}, we have several constraints on the system. We enumerate those constraints:

1. $C_{in} + C_{out} = S$,
2. $L_{mem} = D_{mem} - C_{in}$,
3. $T \cdot D_{ban} > C_e + L_{net}$,
4. $T \cdot D_{ban} < 2\,|S| - D_{mem}$.

We now justify each constraint. Given f concurrently stored $|S|$ bits of S_{i+1} in both D_{mem} and A_{out}, we have $C_{in} + C_{out}$ where C_{in} is the number of bits A_{in} concurrently stores on D and C_{out} is the number of number of bits concurrently stored on A_{out} by definition. Given C_{in} bits of S_{i+1} are stored in D_{mem}, then the maximum number of bits to available A_{in} to store portion of S_i or $w(S_{i-1}, n_{i-1}$ or $w(S_i, n_i)$ is $L_{mem} = D_{mem} - |S|$ by the *Fixed Memory Assumption*. Given C_{out} bits of S_{i+1} bits of S_{i+1} are stored in A_{out}, then $T \cdot D_{ban} > C_{out} + L_{net}$. A_{in} can only access

A_{out} through the communication channels, therefore if A_{in} wants to store C_{out} bits A_{in} must send C_{out} bits to A_{out}. The constraint $T \cdot D_{ban} < 2\,|S| - D_{mem}$ is a requirement of the DABLS system explained in Section 3.1.

We use the constraints to show that the amount of memory A_{in} can use to leak S_i or $w(S_{i-1}, n_{i-1}$ or $w(S_i, n_i)$ is $L =$. We have for time $t > (i+1)T_e$, A_{in} cannot leak more than $L = L_{net}+L_{mem} = D_{mem} - C_{in} + T \cdot D_{ban} - C_{out} = D_{mem} - |S| + T \cdot D_{ban} << S$ bits of S_i.

We have proved both cases, therefore we have given V receives the correct R_i by time $(i+1)T_e$, for time $t > (i+1)T_e$, it is computationally infeasible for A_{in} to leak more than $L = L_{net}+L_{mem} = D_{mem} - |S| + T \cdot D_{ban} << S$ bits of S_i or $w(S_{i-1}, n_{i-1}$ or $w(S_i, n_i)$. By Requirement 3.2.5, A_{in} leaking S_i or $w(S_i, n_i)$ or $w(S_{i-1}, n_{i-1})$ is equivalent, therefore given V receives the correct R_i by time $(i+1)T_e$, for time $t > (i+1)T_e$, A_{in} can leak at most L_{mem} bits of S_i. □

We finish the proof of theorem 3.2.2. By Requirement 3.2.3, A_{out} cannot independently compute bits of S_i.A_{out} learns bits of S_i only if A_{in} leaked those bits. Lemmas 3.2.1, 3.2.2, 3.2.3 encompass all time t, therefore A_{in} can never leak more than $L = L_{mem} + L_{net} << S$ bits of S_i if V receives the correct R_i by time $(i+1)T_e$. We have shown both A_{out} only knows bits of S_i if A_{in} those bits to A_{out} and A_{in} cannot leak more than L bits of S_i if V receives the correct R_i by time $(i+1)T_e$. Therefore we completed theorem 3.2.2.

The proof of theorem 3.2.1 follows from theorem 3.2.2.

3.3 The Pool Update Function f

We present our realization of a concrete implementation of f. Our implementation of f consists of two parts a pseudo-random function and a block selection mechanism. First, we will give an overview of our pseudo-random function. Second, we describe what a block selection mechanism does. Third, we will propose several block selection mechanisms. Fourth, we will perform detailed analysis on some of the block selection methods. Figure 3.3 summarizes the notation used in section.

$\|b\|$	block size in bits
$S[i]$	block i of pool S
E	A block cipher's encryption function
$f(n, S_i)$	Return pool S_{i+1}
$g(n, S, i)$	Recursive function family used to define f
$g_{nS}(i)$	g instantiated with n and S
λ	how many times to compute g_{nS}
N	Number of blocks in S: $\frac{\|S\|}{\|b\|}$
$P(n, bs)$	Non-invertible PRF
$P_n(bs)$	P keyed with n

Figure 3.5: Notation for defining f

3.3.1 Pseudo-random function

We describe our pseudo-random function. We suppose we have a family of non-invertible PRF's P. We use a nonce n to select which particular PRF P_n from that family (Equation (3.1a)). An instance of P_n takes as input k blocks of size $|b|$ bits, and produces one block of output (Equation (3.1b)).

$$P : \{0,1\}^{|n|} \rightarrow P_n \tag{3.1a}$$

$$P_n : \{0,1\}^{k \cdot |b|} \rightarrow \{0,1\}^{|b|} \tag{3.1b}$$

We say that P_n is non-invertible in that knowledge of n, the output, and part of the input does not reveal any more of the input.

One concrete way of implementing a suitable P is similar to the CBC-MAC algorithm. As with CBC-MAC, we use a block-cipher encryption function E in CBC mode with initialization-vector 0, and then output only the last block. Normally, this block could be decrypted to recover the last block of input; we make the function non-invertible by xor-ing the output block with the last block of the input: We propose instantiating P using a CBC-MAC based on a block-cipher E (e.g., AES). Recall that CBC-MAC xors each block of input with the preceding block of cipher text (starting with an IV of 0), and then encrypts it with the block cipher. The output is the last encrypted block. Normally, knowledge of the key would allow the output to be decrypted to recover the last block of

plain text. To avoid this, we xor the final output with the last plain text block.

3.3.2 Generic Block Update

We present a update function using a generic block selection mechanism. We define our pool update function f in terms of a recursive function $g_{n,s}$.

We realize generic f as follows:

$$S[i] = \text{block } i \text{ of } S \tag{3.2a}$$

$$g_{nS}(i) = \begin{cases} S[i] & \text{if } i < N \\ P_n(g_{nS}(i-N)||g_{nS}(BLOCK(k,i)) \dots \\ \quad \dots g_{nS}(BLOCK(1,i))) & \text{if } i \geq N \end{cases} \tag{3.2b}$$

$$f(n,S) = g_{nS}(\lambda) \dots g_{nS}(\lambda + N - 1) \tag{3.2c}$$

We break the pool S into N blocks of size $|b|$ bits, referring to block i of the pool S as $S[i]$ (Equation (3.2a)). The update function can use K dependencies in the update computation. The function BLOCK takes in a pool state, a dependency index and a block index and outputs a function index in the range of $i - N + 1 : i - 1$. The dependency index k has the range of $1 : K$. The recursive function g_{nS} (Equation (3.2b)) takes an index i as input and produces a single block as output. We define $g_{nS}(0)$ to $g_{nS}(N-1)$ to be the input blocks $S[0]$ to $S[N-1]$. Finally, f itself is defined as the last N blocks after computing λ blocks of g_{nS} (Equation (3.2c)). Clearly λ must be at least N so that the output of f is distinct from its input. We further examine the choice of the design parameter λ in Section 3.3.4.

Security of Generic Function

We show that the generic function satisfies most of the requirements present in section 3.2.2

We now show that f satisfies Requirement 3.2.2:

> **f is Non-Invertible**. Given, for all $k < i$, L bits of S_k, nonces n_k, and responses R_k and any S_i, it is computationally infeasible to compute additional bits of any S_k.

All of $f(S, n)$'s output is the output of a non-invertible PRF that depends, directly or indirectly, on every bit of the input pool S. Given that fewer than L bits of S, it is computationally infeasible to compute any additional bits of S.

We now show that f satisfies Requirement 3.2.3:

> **f Needs Complete Pool**. Given, for all $k < i$, L bits of S_k, nonces n_k, and responses R_k and n_i, it is computationally infeasible to compute any bits of $f(S_i, n_i)$ without all of S_i.

Each output block of $f(S_i, n_i)$ can be expressed as a recursion tree of g_{nS}. This recursion tree incorporates every input block of S_i as an input to the non-invertible PRF. Therefore, there is no way to compute any output block of $f(S_i, n_i)$ without knowing the entire input S_i. We now show that f satisfies Requirement 3.2.4:

> **f Cannot Compute Early**. Given, for all $k < i$, L bits of S_k, nonces n_k, and responses R_k and S_{i-1}, It is computationally infeasible to compute any bits of $f(S_i, n_i)$ without n_i

Each output block of $f(S_i, n_i)$ is the output of a PRF from the family P. n_i is needed to know which PRF P_n is to be used. Hence it is infeasible to compute any part $f(S_i, n_i)$ without n_i.

3.3.3 Block Selection Mechanisms

The block selection mechanism selects chooses which recursive dependencies a g_{nS} computation will depend on. Designing the block selection mechanism is interesting because the recursive dependency structure can significantly impact the update function's resilience to the split computation attack and the time space trade-off attack.

CBC-Like

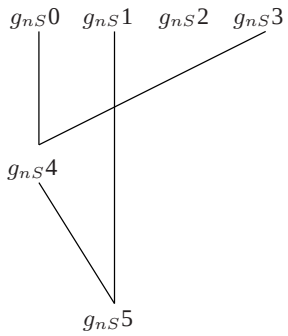

Figure 3.6: Recursion Tree for CBC-like Scheme

We give an overview of the CBC-like block selection mechanism. The CBC-like block selection mechanism selects the k most recent g_{nS} computations as dependencies. Therefore, CBC-like selection mechanism is realized as $BLOCK(k, i) = i - k$. The advantage of the CBC-like block selection mechanism is its easy to analyze. The disadvantage of the CBC-like selection scheme is a update function using the CBC-like selection scheme may need to use more dependencies to be secure.

Figure 3.6 shows a partial example recursion tree that would be generated by the update function using the CBC-like block selection mechanism. In this example the update function uses the CBC-like block selection mechanism to select one dependency per block update. The main purpose of the figure to give a visual representation on how the g_{nS} computations depend on each other when the CBC-like scheme is used as the block selection mechanism. In this figure we see that $g_{nS}4$ depends on $g_{nS}0$ and $g_{nS}3$. $g_{nS}4$ depends on $g_{nS}0$ by definition of our update function. $g_{nS}4$ depends on $g_{nS}3$ by selection by the CBC-like block selection mechanism because $BLOCK(4, 1) = 4-1 = 3$. We show that $g_{nS}5$ depends on $g_{nS}1$ and $g_{nS}4$ to illustrate how the recursion tree would continue.

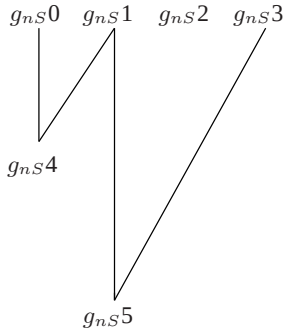

Figure 3.7: Recursion Tree for Pseudo-Random Scheme

Pseudo-Random Scheme

We give a brief description of our pseudo-random block selection mechanism. The pseudo-random scheme select k dependencies by using each value of the last k g_{nS} computations modulo the pool-size. We realize the pseudo-random block selection mechanism as $BLOCK(k,i) = (g_{nS}(i-k) \mod |S|) + i - N$ where $|S|$ is the size of the pool in blocks.

Figure 3.7 shows a partial example recursion tree that would be generated by a update function using the pseudo-random block selection mechanism. In this example the update function uses the pseudo-random block selection mechanism to select one dependency per block update. In this figure we see that $g_{nS}4$ depends on $g_{nS}0$ and $g_{nS}1$. $g_{nS}4$ depends on $g_{nS}0$ by definition of our update function. $g_{nS}4$ depends on $g_{nS}1$ by selection from the pseudo-random block selection scheme. We $g_{nS}5$ and its dependencies to give a sense on how the recursion tree would continue.

Public Permutation Scheme

We give a brief description of the public permutation scheme. The public permutation scheme uses a linear congruential generator to select which g_{nS} computations to use as a dependency. We realize the public permutation scheme as follow as $BLOCK(k,i) = a(k+i) \mod |S| + i - |S|$ where $a = i + (i+1 \mod 2) \mod |S|$. In the public permutation scheme we assume that the pool size is a power of 2.

Figure 3.8 shows a partial example recursion tree that would be generated by the update function

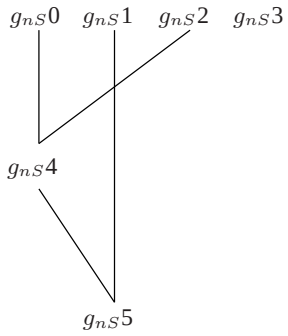

Figure 3.8: Recursion Tree for Public Permutation Scheme

using the public permutation block selection mechanism. In this example the update function uses the public permutation block selection mechanism to select one dependency per block update. In this figure we see that $g_{nS}4$ depends on $g_{nS}0$ and $g_{nS}1$. $g_{nS}4$ depends on $g_{nS}0$ by definition of our update function. $g_{nS}4$ depends on $g_{nS}1$ by random selection from the public permutation block selection scheme. We $g_{nS}5$ and it's dependencies to give a sense on how the recursion tree would continue.

Hoppy Scheme

We give a brief description of the hoppy scheme. The hoppy scheme has two modes. The modes are selected by computing $MODE = \lfloor i/|S| \rfloor \mod 2$. On $MODE = 0$ the hoppy scheme uses the CBC-like function to select dependencies. On $MODE = 1$ the hoppy scheme select blocks by computing the index plus the pool size over 2 modulo the pool size. We realize the hoppy scheme as $BLOCK(k, i) = BLOCK_{MODE}(k, i)$ where $BLOCK_0(k, i) = i - k$ and $BLOCK_1(k - i) = i - (|S|/2)$.

Figure 3.9 and Figure 3.10 shows a partial example recursion tree that would be generated by the update function using the hoppy block selection scheme. Figure 3.9 shows that the hoppy scheme where $MODE = 0$ selects dependencies in the same fashion as the CBC-like scheme. Figure 3.10 shows how the recursion structure is different when $MODE = 1$.

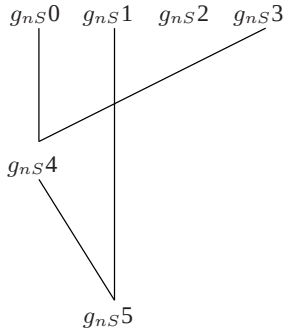

Figure 3.9: Hoppy $MODE = 0$ Recursion Tree

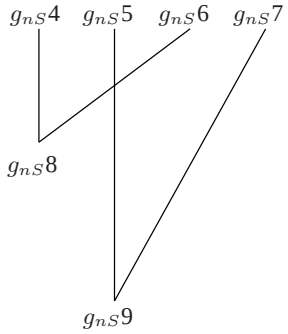

Figure 3.10: Hoppy $MODE = 1$ Recursion Tree

3.3.4 Detailed Analysis

Use Every Block

A common design characteristic of each block selection variant is that every block is a guaranteed to be selected once every $|S|$ block updates We now explain why it would be bad if this was not the case. Suppose the device has just finished computing block update i and the block selection mechanism does not use the block as a dependency during the next $|S|$ block updates, then the device does not need to store the result of block update i because block update i will never ever be used as a dependency. This is due to the fact that an efficient implementation that concurrently stores $|S|$ blocks simultaneously will overwrite the result of block update i with block update $i+|S|$ after $|S|$ block updates. If a device does not need to store the result of a block update i then update will be more vulnerable to the time space trade-off attack. If a block update i does not need to be

stored, then for $|S|$ pool updates after i the pool state is compressible by one block. This would give the attacker one block of extra space to use for memoization for the time space tradeoff attack.

Detailed Security Analysis of CBC-like

We analyze how the security of an update using the CBC-like block selection mechanism changes as the number of dependencies change. We show that when the CBC-like block selection mechanism selects only one additional dependency then the update function is vulnerable to a split computation attack. Then we show that when the update function uses the CBC-like function to select n dependencies then the update function is secure against against both the split computation attack and time space trade off attack.

CBC-like with two dependencies is insecure to the split computation attack. We present the basic idea of the attack. Ain leaks l contiguous blocks in S_i. Then during the update from S_i to S_{i+1} the internal adversary continually leak the dependency of the first block of the contiguous blocks. This would allow the external adversary to update the contiguous blocks from S_i to S_{i+1}.

We present the attack algorithm. We consider a pool update from S to S'. Before the pool update, we assume the attacker starts with a contiguous region of memory leaked starting at index i of l length. During the pool update, the attacker does the following. for $(j = 0 : j < \lambda/|S| : j++)$ The internal adversary A_{in} leaks $g_{nS}(i+j(|S|+1)$. After the pool update, external attacker compute the update using the blocks leaked during the update and the blocks leaked before the update.

We give a step by step example of the split computation attack on the CBC-like scheme. We use the example pool 2 of size 8. We set $L_{BAN} = 4$. We set the pool update parameters to $\lambda = 8$ and $k = 2$. With these parameters $S_0[0] = g_{nS}(0)$ and $S_1[0] = g_{nS}[8]$. The attacker's goal is for the external attacker to learn 7 blocks of S_1 despite $L_{BAN} = 4$. During epoch 0, the internal adversary A_{in} leaks blocks 1,2,3 of S_0. During epoch 1, the internal adversary A_{in} leaks blocks 0,4,5,6 of S_1. After epoch 1, the external adversary A_{out} uses block 0 of S_1 and block 1 of S_0 to compute block 1 of S_1, because by definition $S_1[1] = g_{nS}(9) = P_n(g_{nS}(1)||g_{nS}(8)) = P_n(S_0[1]||S_1[0])$. The external adversary also computes $S_1[2] = g_{nS}(10) = P_n(g_{nS}(2)||g_{nS}(9)) = P_n(S_0[2]||S_1[1])$ and

c	blocks of memory used for memorization
C_{P_n}	cost to compute P_n on each input block
$C(i)$	cost to recompute ith memorization miss

Figure 3.11: Notation for computation cost analysis of f

S	$g_{nS}(0)$	$= S[0]$
	$g_{nS}(1)$	$= S[1]$
	$\ldots$	$\ldots$
	$g_{nS}(N-1)$	$= S[N-1]$
	$g_{nS}(N)$	$= P_n(g_{nS}(0) \ldots g_{nS}(N-1))$
	$g_{nS}(N+1)$	$= P_n(g_{nS}(1) \ldots g_{nS}(N))$
	$\ldots$	$\ldots$
$f(n,S)$	$g_{nS}(\lambda)$	$= P_n(g_{nS}(\lambda-N) \ldots g_{nS}(\lambda-1))$
	$g_{nS}(\lambda+1)$	$= P_n(g_{nS}(\lambda+1-N) \ldots g_{nS}(\lambda))$
	$\ldots$	$\ldots$
	$g_{nS}(\lambda+N-1)$	$= P_n(g_{nS}(\lambda-1) \ldots g_{nS}(\lambda+N-2))$

Figure 3.12: Computation of f via g_{nS}

$S_1[3] = g_{nS}(11) = P_n(g_{nS}(3)||g_{nS}(9)) = P_n(S_0[3]||S_1[2])$. At this point the external adversary A_{out} has learned 7 blocks of S_1. The attack is successful.

We give a proof on how the CBC-like function is secure against both the time space trade-off attack and the split computation attack when the CBC-like function uses n dependencies.

We first demonstrate that the CBC-like function using n dependencies is secure against the time space trade-off attack. We will do this by analyzing the computation cost of computing f using the CBC-like block selection with n dependencies. Our unit of computation in terms of the underlying PRF P_n. We analyze two cases. The first case is where the implementation of f uses less than $|S|$ blocks of memory to memoize results of g_{nS}. We show that for the computation cost when $c < N$ grows *exponentially* with the parameter λ, while the computation cost when $c \geq N$ grows only linearly with λ. Hence, the cost to compute f with fewer than N blocks of memoization cache can be made arbitrarily more expensive than for computation with N blocks of memoization cache. Because all blocks in the memoization cache are output of the non-invertible PRF P_n, this data is not invertible to any part of the input S_i.

Figure 3.3.4 summarizes new terms introduced in this analysis.

Figure 3.3.4 illustrates a computation of f and g_{nS}. An efficient implementation of f begins computing g_{nS} at $i = N$, incrementally computing *forward* the next g_{nS}. To avoid recomputing previous values of g_{nS}, the last N blocks must be stored. When $g_{nS}(i)$ is computed, $g_{nS}(i - N)$ may be overwritten, since it will never be used directly again to compute $g_{nS}(j)$ for $j > i$. This includes overwriting the original input $S = g_{nS}(0) \ldots g_{nS}(N - 1)$.

Computation of $g_{nS}(i)$ requires the N preceding blocks. An efficient implementation that uses c blocks of memoization cache avoids recomputation by always storing the most recent N blocks. The cost to compute f in this manner, using $|S| = N \cdot |b|$ bits of memory to store intermediate blocks, is the number of $g_{nS}(i)$ to compute, times the cost to compute P_n over N blocks, which we define as $N \cdot C_{P_n}$, giving a total cost of $\lambda \cdot N \cdot C_{P_n}$.

An implementation that uses $c < N$ blocks of memoization cache is forced to recursively recompute earlier blocks. Consider an implementation of f that directly computes $f(S, n) = g_{nS}(\lambda) \ldots g_{nS}(\lambda + N - 1)$ one block at a time by recursively computing g_{nS}, using c blocks of memory to memoize results.

The recursion tree for computing the first output block g_{nS} (λ) eventually recurses back to computing $g_{nS}(N)$, which can be computed directly at a cost of $N \cdot C_{P_n}$, and that block may be memoized assuming that $c \geq 1$. The next block that will need to be computed in the recursion tree is $g_{nS}(N+1)$. The memoized block $g_{nS}(N)$ can be used, allowing $g_{nS}(N+1)$ to also be computed for a cost of $N \cdot C_{P_n}$.

The recursion tree is such that each $g_{nS}(i)$ will be computed sequentially in this manner, up until $g_{nS}(N + c)$. That block may be computed for the same cost of $N \cdot C_{P_n}$, but no memory remains to memoize it. We first consider the case that the first c blocks to be computed and memoized are never evicted from the cache. We later argue that no other caching strategy does any better than this one given $c < N$.

Let $C(j)$ be the cost to compute the jth unmemoized block. The cost of computing missing block 0, $g_{nS}(N + c)$, is $C(0) = N \cdot C_{P_n}$, as noted above. To compute missing block 1, missing block 0 must first be recomputed, giving a cost $C(1) = C(0) + N \cdot C_{P_n} = 2 \cdot N \cdot C_{P_n}$. To compute

missing block 2, first block 0 must be recomputed, then block 1 must be recomputed. Note that to recompute block 1, block 0 must be recomputed *again*; there is no memory remaining to save that block. Hence $C(2) = C(0) + C(1) + N \cdot C_{P_n} = 4 \cdot N \cdot C_{P_n}$.

In general, to compute $g_{nS}(i)$ for $i > (N + c)$, the recursion tree is structured such that *all* $i - N - c$ unmemoized blocks must be sequentially (re)-computed. The cost to compute $g_{nS}(i)$ for $i > (N + c)$ is $C(i - N - c)$. We initially define C in Equation 3.3:

$$C(i) = \begin{cases} N \cdot C_{P_n} & \text{if } i = 0 \\ N \cdot C_{P_n} + \sum_{j=0}^{j=i-1} C(j) & \text{if } i > 0 \end{cases} \tag{3.3}$$

We can find the closed form by noting that for $i \geq 2$, $C(i-1) = N \cdot C_{P_n} + \sum_{j=0}^{j=i-2} C(j)$. Therefore, for $i \geq 2$:

$$\begin{aligned} C(i) &= N \cdot C_{P_n} + \sum_{j=0}^{j=i-1} C(j) \\ &= N \cdot C_{P_n} + C(i-1) + \sum_{j=0}^{j=i-2} C(j) \\ &= C(i-1) + (N \cdot C_{P_n} + \sum_{j=0}^{j=i-2} C(j)) \\ &= C(i-1) + C(i-1) \\ &= 2 \cdot C(i-1) \end{aligned}$$

Combined with the base case that $C(0) = N \cdot C_{P_n}$, this gives us the closed form for C (Equation (3.4)):

$$C(i) = N \cdot 2^i \cdot C_{P_n} \tag{3.4}$$

Recall that this cost is based on the memoization strategy of storing the first c computed blocks,

and then never evicting those from the cache. Can the adversary do better with a cleverer memoization strategy? We argue that he cannot. When computing $g_{nS}(i)$ for $i > N + c$, *at least* $i - N - c$ previous blocks are not stored in the memoization cache. By the above analysis, it doesn't matter at all *which* blocks are missing from the cache and must be recomputed, only how many. Further, since computation of g_{nS} involves a unique prefix to P_n, one partially computed block cannot be used to help compute other blocks.

At last we explicitly prove the requirement. When at least $|S|$ bits (N blocks) are used to store results of g_{nS}, the cost to compute f is $\lambda \cdot N \cdot C_{P_n}$. When $|S|-1$ bits are used, this leaves $c = N-1$ blocks to cache intermediate blocks. The total cost is equal to the cost to compute the first $c = N-1$ blocks, plus the cost to compute the remaining of the final N blocks. This cost is given below. [2]

$$N \cdot (2^{\lambda - N + 1} + N - 2 - max(2^{\lambda - 2N + 1} - 1, 0)) \cdot C_{P_n}$$

To satisfy the requirement that f may be computed in less than T_{up} if and only if at least $|S|$ bits of memory are overwritten, we must satisfy the relation:

$$\begin{aligned} & \lambda \cdot N \cdot C_{P_n} \\ < \;& T_{up} \\ < \;& N \cdot (2^{\lambda - N + 1} + N - 2 \\ & - max(2^{\lambda - 2N + 1} - 1), 0) \cdot C_{P_n} \end{aligned} \tag{3.5}$$

Since the cost to compute f when using $|N|$ bits to store output blocks of P_n is polynomial in N and λ, and the cost to compute f when using fewer bits is *exponential* in λ, it is straightforward to select parameters so that the sub-$|S|$-bit implementation is arbitrarily more expensive than the

[2] Note that there is no need to preemptively compute the blocks in between the last memoized block and the first output block. The max term accounts for when there are such "middle" blocks that do not need to be explicitly computed except as part of the computation of later blocks.

$|S|$-bit implementation. For example, setting $\lambda = N + 128$ where $N > 128$ gives us:

$$\begin{aligned} & (N^2 + 128N) \cdot C_{P_n} \\ < \quad & T_{up} \\ < \quad & N \cdot (2^{129} + N - 2) \cdot C_{P_n} \end{aligned} \tag{3.6}$$

The other part of the requirement is that the $|S|$ stored bits must not be invertible to any part of the input pool. Since the stored blocks are the output of the non-invertible PRF P_n, this requirement is trivially met.

We now show that f satisfies Requirement 3.2.5:

> **Leakage Equality for f**. Given, for all $k < i$, l bits of S_k, nonces n_k, and responses R_k and $L - Z$ bits of S_i, Z bits of outputs from any function $w(S_i, n_i)$ computable by A_{in} cannot be used to compute more than Z bits of S_i or S_{i+1} where $S_{i+1} = f(S_i, n_i)$.

This property trivially holds when $L + Z \geq |S|$, because there is then no unknown information left to compute.

We therefore consider only the case where $L + Z < |S|$. We show that no part of $f(S_i, n_i)$ can be computed with less than $|S|$ bits of knowledge. Since all parts of the output of $f(S_i, n_i)$ are output blocks of g_{nS}, computation of some part of the output of f implies computation of some $g_{nS}i$. We next show that computation of any $g_{nS}i$ is infeasible when less than $|S|$ bits of information has leaked.

$g_{nS}(i)$ for $0 \leq i \leq N - 1$ are the input blocks $S[0]$ to $S[N - 1]$, by definition of g_{nS} (Equation (3.2c)). Computation of more of these input blocks than the size of the leaked information $L + Z$ would imply compression of that data. Since the input is pseudo random, no such function w can exist.

Block $g_{nS}(N)$ is the first computed block. By the definition of g_{nS}, computation of this block requires knowledge of all $|S|$ bits of its input, which is exactly S_i. Since we have shown that no

more than $L + Z$ bits of that information can be leaked or computed, and $L + Z < |S|$, there is not enough information to compute $g_{nS}(N)$.

Block $g_{nS}(N + 1)$ is the next computed block. Computation of that block requires $g_{nS}(N)$ (which is size $|b|$), and $|S| - |b|$ blocks of the original input. The total size of that information is again $|S|$. We have already shown that less than $|S|$ bits of that information could have leaked directly or been computed from other leaked information. Hence, $g_{nS}(N + 1)$ is also impossible to compute.

This continues recursively for all blocks. Since no g_{nS} can be computed from less than $|S|$ bits of leaked information, then it is impossible for leakage of any Z bits of information to reveal more than Z bits of $f(S_i, n_i)$.

Hoppy Vulnerable to Split Computation Attack

We present a split computation attack against the hoppy scheme. The idea of the attack is to update two regions when the hoppy scheme in $MODE = 0$ such that when $MODE = 1$ the internal adversary does not have to leak anything for the external adversary to compute anything. We present an algorithm for the attack. First the attacker leaks two regions of contiguous blocks at indexes i and k such that $i = k+|S|/2 \mod |S|$. for $(j = 0 : j < \lambda/|S| : j++)$ The internal adversary A_{in} leaks $g_{nS}(i + j(|S| + 1)$ and $g_{nS}(k + j(|S| + 1))$. After the pool update, external attacker compute the update using the blocks leaked during the update and the blocks leaked before the update.

We give a step by step example of the split computation attack against the hoppy scheme. We set the parameters to $|S| = 8$, $\lambda = 16$, and $L = 4$. The attackers goal is to learn 6 blocks of S_1 despite being able to leaks only 4 blocks per epoch. During epoch 0, the attacker leaks $S_0(3)$ and $S_0(7)$. During the pool update, we compute $MODE = \lfloor 8/4 \rfloor \mod 2 = 0$. By our attack algorithm, the internal adversary A_{in} will leak $g_{nS}(10)$ and $g_{nS}(14)$. $S_0(6)$. The external adversary can compute will use $g_{nS}(10)$, $g_{nS}(14)$, $S_0(3)$ and $S_0(7)$ to compute $g_{nS}(11)$ and $g_{nS}(15)$, because $g_{nS}(11) = P_n(g_{nS}(3)||g_{nS}(10))$ and $g_{nS}(15) = P_n(g_{nS}(7)||g_{nS}(14)$. Next, the external adversary will use $g_{nS}(10)$ and $g_{nS}(14)$ to compute $g_{nS}(18)$. We compute block $BLOCK_1(1, 18) =$

$18 - (8/2) = 14$. We have $g_{nS}(18) = P_n(g_{nS}(10)||g_{nS}(14))$. Similarly, we have $g_{nS}(19) = P_n(g_{nS}(11)||g_{nS}(15))$. The external attack can also compute $g_{nS}(22) = P_n(g_{nS}(14)||g_{nS}(18))$ and $g_{nS}(23) = P_n(g_{nS}(15)||g_{nS}(19))$. The external attacker now knows four blocks of S_1 because $S_1[2] = g_{nS}(18)$, $S_1[3] = g_{nS}(19)$, $S_1[6] = g_{nS}(22)$, and $S_1[7] = g_{nS}(23)$. Due the epochs 0 and 1 overlapping during the update process, the internal adversary can leak 2 additional blocks of S_1. After S_1 the external adversary has learned 6 blocks of S_1 despite L being 4. The attack is successful.

Compressible Analysis

We present some analysis on how to adjust the scheme parameters if the pool is compressible. If the pool is x percent compressible then $L_{mem} + L_{ban} \leq L$ where $L_{mem} = D_{mem} + (1 - x)|S|$. We need to adjust the safety parameter because a compressible pool gives the attacker space when the attacker compresses the pool.

3.4 Verifiable Malware-Free State

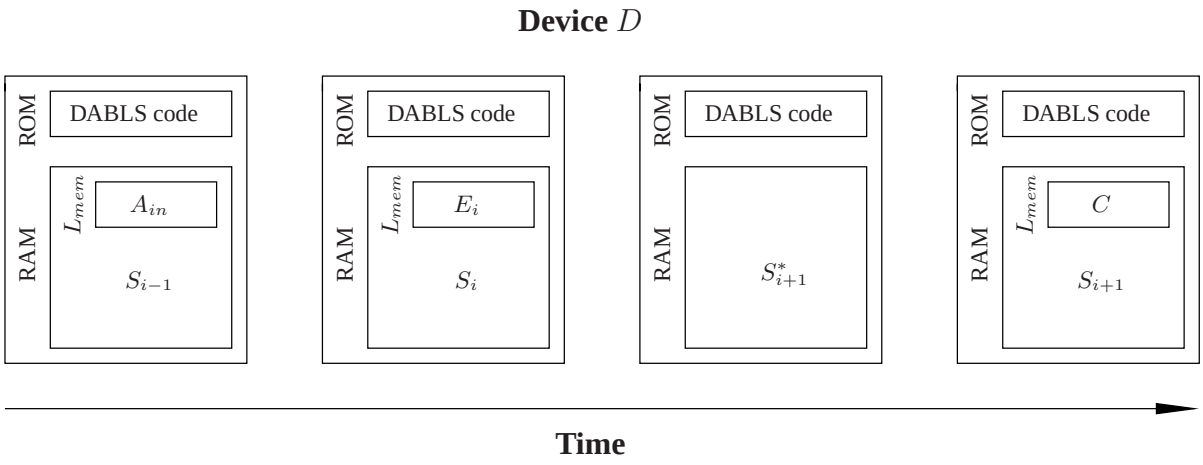

Figure 3.13: Achieving malware-free state at epoch $i + 1$.

In this section we describe a simple protocol that extends DABLS to attempt the removal of

malware (the adversary A_{in}) from device memory, and the loading of intended application code C.

Suppose that we would like device D to run some application code represented by C. We must have $|C| \leq L_{mem}$, otherwise there will not be enough space to store both the pool S and the code C in D_{mem}. Malware is erased from D if D_{mem} does not contain any bits of A_{in}.

Our protocol contains two stages: (1) verifiable remote erasure of malware A_{in}, and (2) loading of the application code, and is implemented in device D's ROM. A small amount of D_{mem} is also required as a workspace Q for the ROM code to execute, which we detail below. Figure 4.9 shows how the memory on D evolves during the protocol. In epoch i:

1. Verifier V sends nonce n_i and entropy E_i to device D, where $|E_i| = D_{mem} - |S| - |Q|$, and Q represents the minimum scratch space necessary for the ROM code (functions f and y) to execute. For example, Q may contain the runtime context for a keyed MAC and a block cipher (see Section 4.1).

2. Device D overwrites L_{mem} with entropy E_i and hence the malware A_{in} is overwritten in D_{mem}. D then computes $f(E_i||S_i, n_i) = S^*_{i+1}$. Note that $|S^*_{i+1}|$ is larger than a typical S (recall that $|S| = |D_{mem}| - |L_{mem}|$), since its domain and range include $|S| + |E|$ bits. D sends $R_i = y(S^*_{i+1})$ to V.

3. Verifier V checks that D's response is correct and timely and, if so, it sends application code $\{C, HMAC_{S_{i+1}}(C)\}$ to device D, where $|C| \leq L_{mem}$; otherwise, V initiates D's reset. Note that S_{i+1}, as shown in Figure 4.9, is simply S^*_{i+1} with $|C|$ bits overwritten with C, and that this all takes place during a single epoch.

4. Device D authenticates C using the $HMAC$ and S_{i+1}, and then overlays L_{mem} with application code C, and responds with $\{M, HMAC_{S_{i+1}}(M)\}$ to V where M is a message confirming D has received C. Verifier V checks that D's response is correct and timely, and, if so, V considers code C to be activated on device D in a malware-free environment; otherwise V attempts D's reset.

The above extension of the DABLS protocol requires that enough ROM is available in device D to enable the execution of the protocol.

Note that the verifier does not have to run the protocol for achieving a malware-free state on device D in every epoch. During epochs where application code C needs to receive and execute management commands, verifier V uses the basic DABLS protocol presented in Section 3 to authenticate device D.

Lemma 3.4.1. *Given that V receives correct and timely response $C, HMAC_{S_{i+1}}(C)$ it is computationally infeasible for A_{in} to prevent the complete loading of application code C in D_{mem}, i.e., the device D's memory content $D_{mem} = C||S_{i+1}$*

Proof. We outline our proof as follows. Suppose A_{in} occupies $a > 0$ bits in D_{mem} and yet V receives the correct, timely response from D, at epoch $i + 1$. If D_{mem} contains bits of malware A_{in} at state $i + 1$, D's response to verifier V's challenge could not have been correct and timely for the new memory pool $D_{mem} = E_i||S_i||Q$. In this case verifier V would have attempted D's reset. Similarly, the correct, timely response received after $\{C, HMAC_{S_{i+1}}(C)\}$ is sent implies that C was correctly loaded in a timely manner. □

3.5 Device Initialization and Reset

3.5.1 Device Initialization

A device's secret pool S comprises its identity. The initial pool, S_0, must be injected before the device can be deployed in a real network and become the subject of adversary attacks. For example, initial pseudo-random pool S_0 could be injected by the manufacturer, or may be "imprinting" on the device by a technician, at installation time; e.g., a "smart meter" may have S_0 is injected before it is initially connected to the power grid.

The verifier (e.g., a process run by an electric utility) requests and receives periodic heartbeat messages from the device that confirm that the device's pool is being updated in accordance with

our protocols. If a response to a heartbeat request is ever incorrect or misses its deadline (e.g., fails to arrive at all), then the state of the device software is considered suspect. Malware may be present on the device.

3.5.2 Device Recovery and Reset

When a device's heartbeat fails correct and timely verification, we initiate another protocol to attempt to recover the device. It is always a possibility that malware refuses to respond to these messages or perform the necessary actions. In this case, it is necessary to send a technician to repair or replace the device. However, malware that behaves thusly will be detected with overwhelming probability. Further, an adversary may prefer to have his malware wiped from the device, instead of attracting undue attention.

If malware is not in permanent control of the device, e.g., it has only hooked an API or interrupt service routine associated with billing, then our protocol for reaching a malware-free device state will be successful. If so, the device is considered to have had its application code and pseudo random pool reinitialized to a known-good state (i.e., S_{i+1} in Figure 4.9), and normal operations can resume. Physical intervention becomes necessary only if this reset protocol fails.

3.5.3 Bootstrapping Additional Applications

If malware is successfully removed from the device, then we have the opportunity to bootstrap any of a number of standard cryptographic libraries and protocols. The device's current secret pool, S_i, represents a shared secret with the verifier. This is sufficient to derive keys for almost any standard cryptographic protocol. Thus, we are able to build from device authentic identity and malware-free state, to a known-good shared secret between devices, to arbitrary cryptographic protocols. Furthermore, achieving a malware-free state complements new cryptographic constructs that are intended to be resilient to continual key leakage via side-channel attacks; viz., the next section.

Chapter 4

Evaluation

4.1 Parameter Exploration

Here we explore practical parameters of epoch duration (Figure 4.1), device memory and pool size (Figure 4.2), and device bandwidth (Figure 4.3). To aid in intuition, we also offer some parameters of real-world wireless network interfaces (Figure 4.4).

We let $L_{mem} = D_{mem} - |S|$ and $L_{net} = T \cdot D_{ban}$.

T (seconds)	$\|S\|$ (MB)	L_{net} (MB) $<$	L_{mem} (MB) $<$
1	0.52	0.03	0.48
2	0.53	0.06	0.47
3	0.55	0.09	0.45
4	0.56	0.12	0.44
5	0.58	0.15	0.42
6	0.59	0.18	0.41
7	0.61	0.21	0.39
8	0.62	0.24	0.38
9	0.64	0.27	0.36
10	0.65	0.3	0.35
20	0.8	0.6	0.2
30	0.95	0.9	0.05

Figure 4.1: Implications of varying epoch duration T. We fix the device memory D_{mem} at 1 MB, D_{ban} bandwidth per second as 0.03 MB/s. The L_{net} and L_{mem} values shown here are the upper bounds on allowable leakage.

D_{mem} (MB)	$\|S\|$ (MB)	L_{net} (MB) $<$	L_{mem} (MB) $<$
64	52	40	12
128	84	40	44
256	148	40	108
512	276	40	236
1024	532	40	492
2048	1044	40	1004
4096	2068	40	2028

Figure 4.2: Implications of varying device memory D_{mem}. T is fixed at 400 seconds. D_{ban} is fixed at 0.1 MB/s. The L_{net} and L_{mem} values shown here are the upper bounds on allowable leakage.

D_{ban} (MB/s)	$\|S\|$ (MB)	L_{net} (MB) $<$	L_{mem} (MB) $<$
0.002	8.1	0.2	7.9
0.004	8.2	0.5	7.8
0.040	10.4	4.8	5.6
0.100	14.0	12.0	2.0
0.130	15.5	15.0	0.5

Figure 4.3: Implications of varying device bandwidth D_{ban}. T is fixed at 200 seconds. D_{mem} is fixed at 16 MB. L_{net} is deterministically calculated from D_{ban}. The L_{net} and L_{mem} values shown here are the upper bounds on allowable leakage.

4.2 Implementation

We implemented a prototype of the pool update function. We first describe our system setup followed by the pool update function's implementation details.

Our testbed consists of two systems connected via a wireless Internet connection: a server operating as the pool update verifier, and a client as the pool update prover.

The server is a Intel Centrino system with a 2.4 GHz Dual-core CPU, with 3 GB RAM and

Protocol	Bandwidth (MB/s)
Bluetooth 1.2	0.09
Bluetooth 2.0 + EDR (practical)	0.26
Bluetooth 2.0 + EDR (nominal)	0.38
Bluetooth 3.0 HS	3.00
Zigbee 2.4 GHz	0.03
Zigbee 915 MHz	0.004
Zigbee 868 MHz	0.002

Figure 4.4: Maximum attainable bandwidth of common wireless data communication media.

Program 1 generic-up

Receive $NONCE$ From Server
`generic-up`($NONCE$)
Send SHA-1(POOL) To Server

150 GB of hard disk running the server code and Ubuntu 9.10. We run our client on 2 different platforms. The first platform is a real time system model RTSM of a ARM 1176 in the ARM Real View Debugger. The second platform is an ARM Versatile Express PB 1176 Development Board. The ARM Development Board uses a ARM 1176 processor, 128 MB RAM, 8 MB PSRAM, 2 GB sd card, and 2x64 NOR FLASH. Our current implementation is written in C. Our code uses the polar-ssl library. Our code runs in user mode. Our code has three main components the `setup`, `generic-up`, and `block`. The `setup` component is responsible for reading the nonce from the server and communicating the final result to the server. The `generic-up` component is responsible for computing the non-invertible PRF and updating the pool. The `block` function is responsible for copying k blocks from the pool to a buffer. How the `block` function selects those blocks depends on which block selection strategy it implements. We implemented the CBC-like, pseudo-random, public permutation and hoppy block selection strategies.

We now describe the implementation of our pool update function. The first program we present is `setup`. The second program we present is `generic-up`. The next set of programs, we present are implementations of the different block selection strategies. Implementing several different block selection strategies is useful because different block selection strategies may have significantly different performance and security characteristics.

The `setup` program (psuedo-code shown in Program 1 is triggered when the verifier sends a nonce to begin the pool update process. First, the setup program calls the `generic-up` program to update the pool. Then the setup program computes the SHA-1 hash of the pool and communicates the result to the server.

The `generic-up` program (pseudo-code shown in Program 2 is triggered when the `setup`

Program 2 generic-up

Parameters: $NONCE$
  for $i = 0 \rightarrow \lambda$ **do**
    index $= i \mod |S|$

    for $j = 0 \rightarrow K$ **do**
      DEPARRAY$[j]$ =POOL[BLOCK(k, i)]
    end for
    ANS $= AES - ENC - CBC_{NONCE}$(POOL[index]||DEPARRAY)
    ANS $=$ ANS $\oplus$ POOL[BLOCK$(1, i)$
    POOL[index] =ANS
  end for

program calls it to update POOL using the nonce $NONCE$. The setup program computes λ block updates. Each block update consists of four steps. The first step is that the pool computes the index of the POOL block that is going to be updated. The second step is finding the k dependencies used for $i - th$ block update by calling the `block` function k times. The third step is computing the non-invertible PRF by calling AES-ENC-CBC on the pool block at $index$ index concatenated with the k dependencies found in the second step. Then the `generic-up` function XORs the result of the AES-ENC-CBC computation with the first dependency found by the `block` function. The fourth step is the pool block at index $index$ is updated with the result of the computation described in step three. The fourth step is important because it gives us the invariant that the $|S|$ most recent updates are stored in POOL. More specifically we a guaranteed that if the current block update is i, then $POOL[x \mod |S|] = g_{nS}(x)$ for $x \in i - n...i$.

We now give pseudo code for each of the block selection schemes. The block selection scheme we chose to implement are the CBC-like scheme, the hoppy scheme, the public permutation scheme and the pseudo-random scheme. Each program is trigger when the `generic-up` programs want to compute which k dependencies to use for a specific block update.

The CBC-like function selects the k most recent results as the k dependencies. CBC-like function accomplishes this by returning $i-k \mod |S|$ as its block selection. The CBC-like function has the simplest implementation of our block update mechanisms. The CBC-like block select algorithm

Program 3 CBC-like

Parameters: k and i

return $i - k \mod |S|$

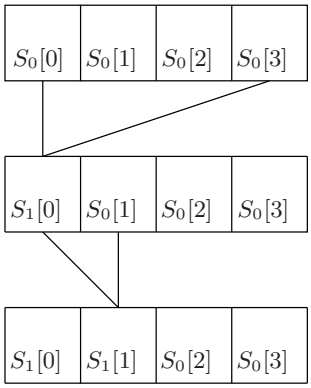

Figure 4.5: CBC-like Block Selection Implementation

shown assumes an efficient pool update implementation therefore it returns the index in memory of which block to uses rather than the g_{nS} index.

Figure 4.5 shows an example update in a efficient implementation of the update function using the CBC-like function implementation. In this example, the pool size is four and number of updates computed in this example is four. The update function is efficient because it will use four blocks of memory for computation. The figure illustrates which blocks in memory are selected in block updates and how memory changes after a block update. We see that to update index 0 from epoch 0 to epoch 1 it depends on block 0 and block 3. We also see that value at index 0 is changed from $S_0[0]$ to $S_1[0]$ immediately, because an efficient implementation writes the results to memory as soon as it computes them to reuse later. We also show the same process for updating index 1.

The pseudo-random block selection function chooses k pseudo-random dependencies. The

Program 4 Pseudo-random

Parameters: k and i

return $POOL[i - k \mod |S|] \mod |S|$

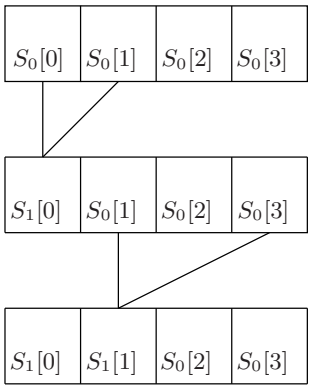

Figure 4.6: Pseudo-random Block Selection Implementation

Program 5 Public Permutation

Parameters: k and i

$a = i + (i + 1 \mod 2) \mod |S|$

$returnak \mod |S|$

pseudo-random block selection function uses the value of the last k blocks modulo the poolsize to pseudo randomly pick k dependencies. This method chooses blocks pseudo-randomly because value of the last k blocks are the output of a pseudo-random function.

Figure 4.5 shows an example update in a efficient implementation of the update function using the CBC-like function implementation. In this example, the pool size is four and number of updates computed in this example is four. The update function is efficient because it will use four blocks of memory for computation. The figure illustrates which blocks in memory are selected in block updates and how memory changes after a block update. We see that to update index 0 from epoch 0 to epoch 1 it depends on block 0 and block 3. We also see that value at index 0 is changed from $S_0[0]$ to $S_1[0]$ immediately, because an efficient implementation writes the results to memory as soon as it computes them to reuse later. We also show the same process for updating index 1.

The public permutation scheme selects k dependencies. The pseudo-random block selection function uses a generator to generate k dependencies.

The hoppy scheme switches between two modes to select blocks. The hoppy scheme only

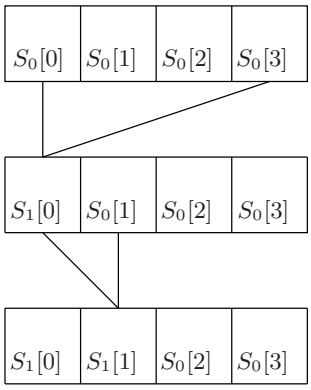

Figure 4.7: Public Permutation Block Selection Implementation

Program 6 Hoppy

Parameters: i

$MODE = \lfloor i/|S| \rfloor \mod 2$

if $MODE == 0$ **then**

return $i - 1 \mod |S|$

else

return $(i - |S|/2)/mod|S|$

end if

selects one dependency. When $MODE == 0$ the dependency returned is the most recent dependency. When $MODE == 1$ the dependency returned is the dependency half away.

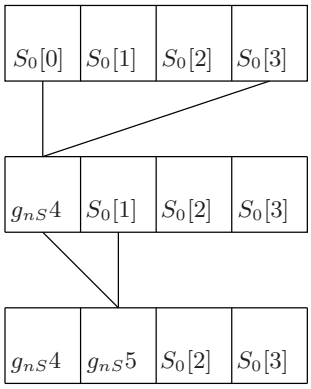

Figure 4.8: Hoppy $MODE = 0$ Block Selection Implementation

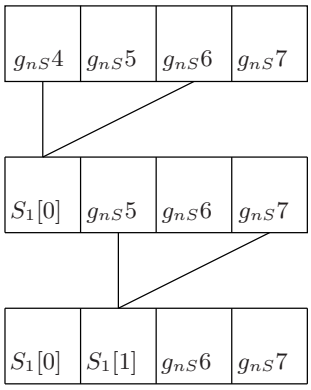

Figure 4.9: Hoppy $MODE = 1$ Block Selection Implementation

Program 7 generic-up

```
Parameters: BLOCK and FRESH
  for i = 0 → λ do
    cost = FRESH[i mod |S|]
    for j = 0 → K do
      rand =randint(0, |S| − 1)
      cost =FRESH[rand]+cost
    end for
    cost = K−cost
    if cost ≤ BLOCK then
      BLOCK = BLOCK - cost
    else
      FRESH[i mod |S|] = 0
      numfresh = numfresh - 1
      if numfresh == 0 then
        return [i, 0]
      end if
    end if
  end for
  return [i, numfresh]
```

4.3 Attack Simulation

4.3.1 Split Computation Attack

Program 7 shows the pseudo-code for a simulation of an split computation attack against the

update function using the pseudo-random block selection scheme. The simulation uses an array that signifies current set of fresh intermediate state that the external adversary has. For every block update within the pool update the simulation picks random dependencies, if the external attacker has all of the dependencies or can leak enough blocks then the external attacker maintains a fresh block otherwise the block is marked as not fresh anymore.

We explain why we think this simulation represents a reasonable attack against the split computation attack scheme. We present several observations that support our claim that our split computation attack is resonable. The first observation is that the attacker's advantage in the split computation attack against an update from epoch i to epoch $i+1$ is how many blocks the external attacker is able to update from i to $i+1$ using intermediate blocks. The second observation is that the attacker can do no better than to break even on state that the attacker leaks during the pool update for example if the attacker leaks three blocks of state in between epoch i and epoch $i+1$ then the attacker can learn no more than three blocks of $i+1$. The third observation, is if the attacker is unable to keep a block fresh during a block update, then the block is useless to the attacker. From these three observations we see that a reasonable strategy is for the attacker to always keep blocks fresh if it can. A limitation of this analysis is that some block may be much more valuable than other blocks due to the random dependencies because blocks that are used as dependencies are more valuable.

We run two simulations to measure the effectiveness of the attack. One simulation measures the performance of the attack as we increase the number of dependencies. The other simulation changes how many blocks the attacker can leak versus how many blocks the attacker can start out with. We measure the effectiveness of the split computation attack by measuring the number of block computations on average where the attacker has at least 1 fresh block. As the number of blocks block computations where the attacker has at least 1 fresh block decreases the effectiveness of the attack decreases because when the attacker has no fresh blocks then the attack is a failure.

We ran a simulation to test how our split computation attack performs as we increase the number of depends. We expect that increasing the number of depends will drastically decrease the performance of the attack. We set the pool size to 128 blocks. The attacker starts with 64 blocks and the

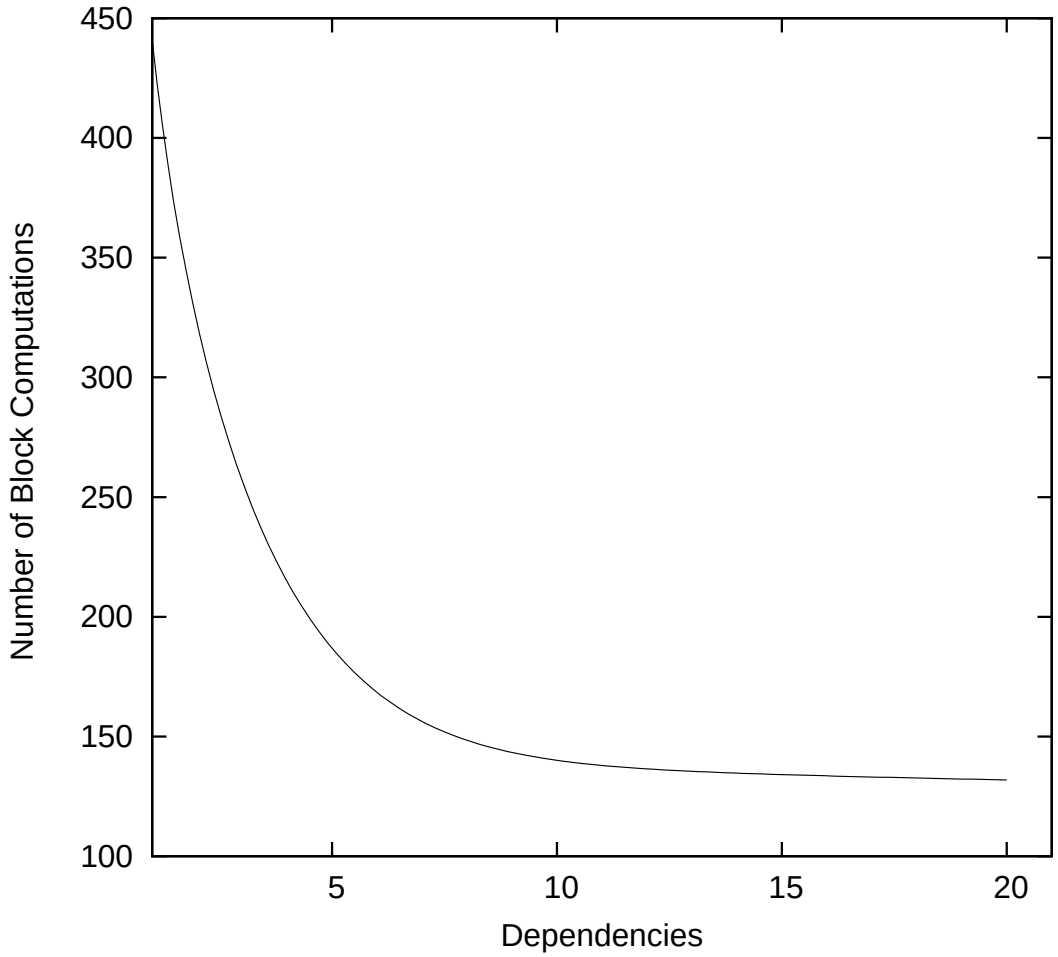

Figure 4.10: Pool Size Scalability

attacker can leak an additional 40. We set the number of block computations to 512. We change the dependencies from 1 to 20. Figure 4.10 shows the result of the simulation. The results show that as the dependencies increase the number of block updates where the attacker has at least one fresh block decreases, therefore increasing the security against the split computation attack. This was expected. This simulation shows that increasing the number of dependencies a good way to increase the difficulty of performing our split computation attack.

We ran a simulation to test how our split computation attack changes as we change the number of blocks the attacker starts with versus the number of blocks the attacker can leak. We set the pool

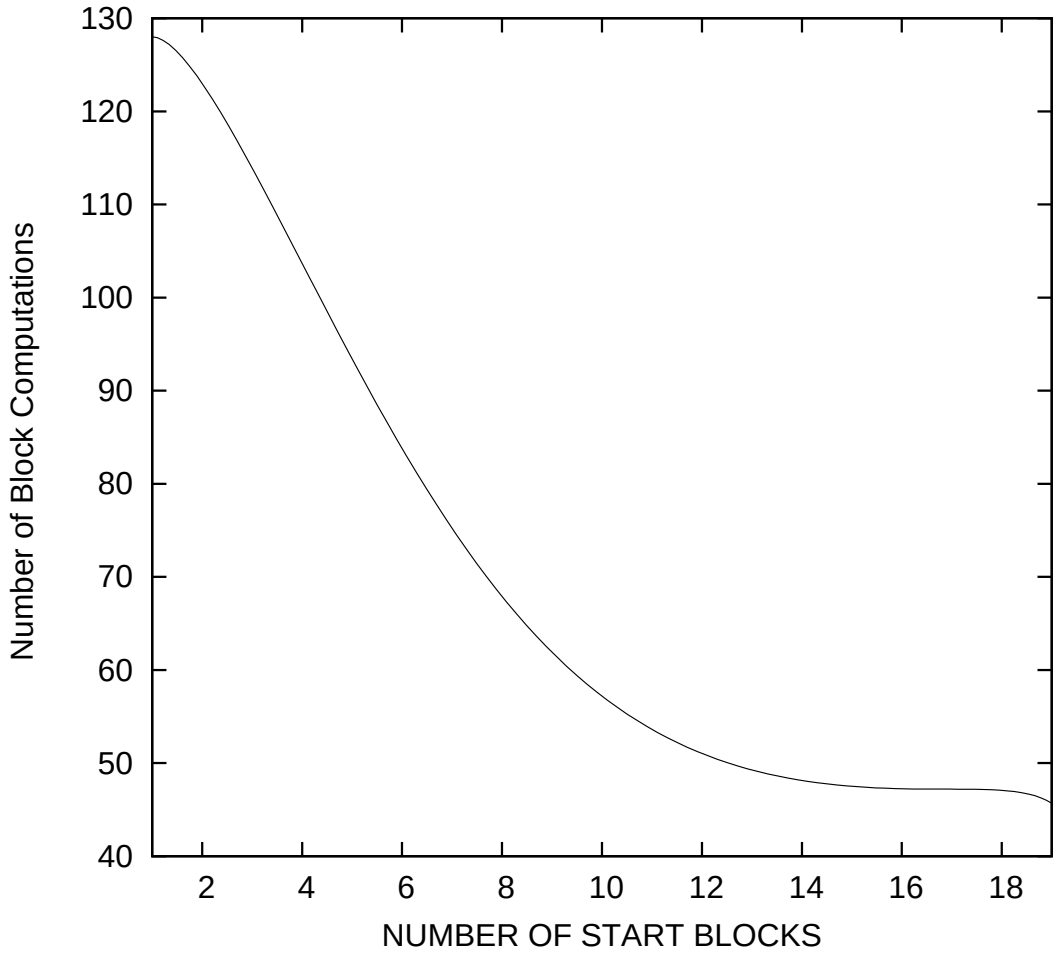

Figure 4.11: Split Simulation Start Blocks

size to 32 blocks. The attacker has 24 blocks total that can be leaked for started with. The number of dependencies is 2. We set the number of block computations to 128. Figure 4.11 shows the result of the simulation. The results show that changing the split between the starting blocks and how many blocks can be leaked does not change the performance of the attack too much. The splits where attacker has low starting blocks and high number blocks to leak during the update look better than they are because the attacker is inefficiently spending block leaks to update a very small amount of blocks.

Program 8 generic-up

```
Parameters: COST
  for i = 0 → λ do
    index = i mod |S|
    CURBLOCK = COST[index]
    if CURBLOCK ≠ 0 then
      if CURBLOCK == 1 then
        COST[index] = K + 1
      else
        COST[index] = K + 1 + CURBLOCK
      end if
      for k = 0 → K do
        rand = randInt(0,n-1)
        dep = COST[rand]
        if dep == 0 then
          COST[rand] = 1
        else
          if dep ≥ K+1 then
            COST[index] = COST[index] + dep
          end if
        end if
      end for
    end if
  end for
  return COST
```

4.3.2 Time Space Trade-off Attack

Program 8 shows pseudo code for a simulation of the time space trade off attack against a pool update using the psuedo random block selection mechanism. The simulation uses a cost array to track the cost of computation changes during the attack. The simulation updates the cost array as it simulates each block update.

We explain why our attack is a reasonable implementation of a time space trade-off attack. We think that a good time space trade-off attack will uses only pool size space, because if there is extra space on the device the attacker can just store old pools without having to perform the time-space trade off attack. We make the observation that the attacker must keep at least 1 block from every

index in memory, because if the attack does not keep blocks from a index then the attacker won't able to compute the correct result. The attacker must keep enough information to recompute all blocks that depend on stale blocks it keeps in memory. This requirement is interesting because if an attacker keeps a block A at index i back, then the block B at index k that is used to update block A to A' at index i will also have to be frozen otherwise the attacker will not able to recompute A'. Therefore, the attacker has little choice in how it memoizes blocks. Therefore, we believe the best time space trade-off attack is for the to memoize as much as possible while still operating under these constraints. A limitation of this analysis is that some block may be much more valuable than other blocks due to the random dependencies because blocks that are used as dependencies are more valuable.

We ran simulations to test the effectiveness of the attack. In the first simulation we measure the effectiveness of the time space trade off attack as we change the number of dependencies. In the second simulation we measure the effectiveness of the attack as we change the number of block computations. The metric for determining effectiveness is the ratio between the attacker cost and the normal cost. We expect that increasing the number of dependencies would make the time space trade-off more expensive for the attacker than increasing the number of block computations.

We ran a simulation to test how our attack performs as we increase the number of dependencies. We expect that increasing the number of dependencies will drastically increase the cost of the attack. We set the pool size to 128 blocks. We set the number of block computations to 512. We set the number of blocks the attacker tries to save to 1. We change the dependencies from 1 to 20. Figure 4.12 shows the result of the simulation. The results show that increasing dependencies is effective at increase the cost of our time trade off attack implementation namely comparing 1 and 6 dependencies, it will take the normal computation six times longer with 6 dependencies however the attacker will take 123844 times to time space trade off against 6 dependencies, versus only 2 times as long with one dependencies.

We ran a simulation to test how our attack performs as we increase the number of rounds. We expect that increasing the number of rounds will drastically increase the cost of the attack. We set

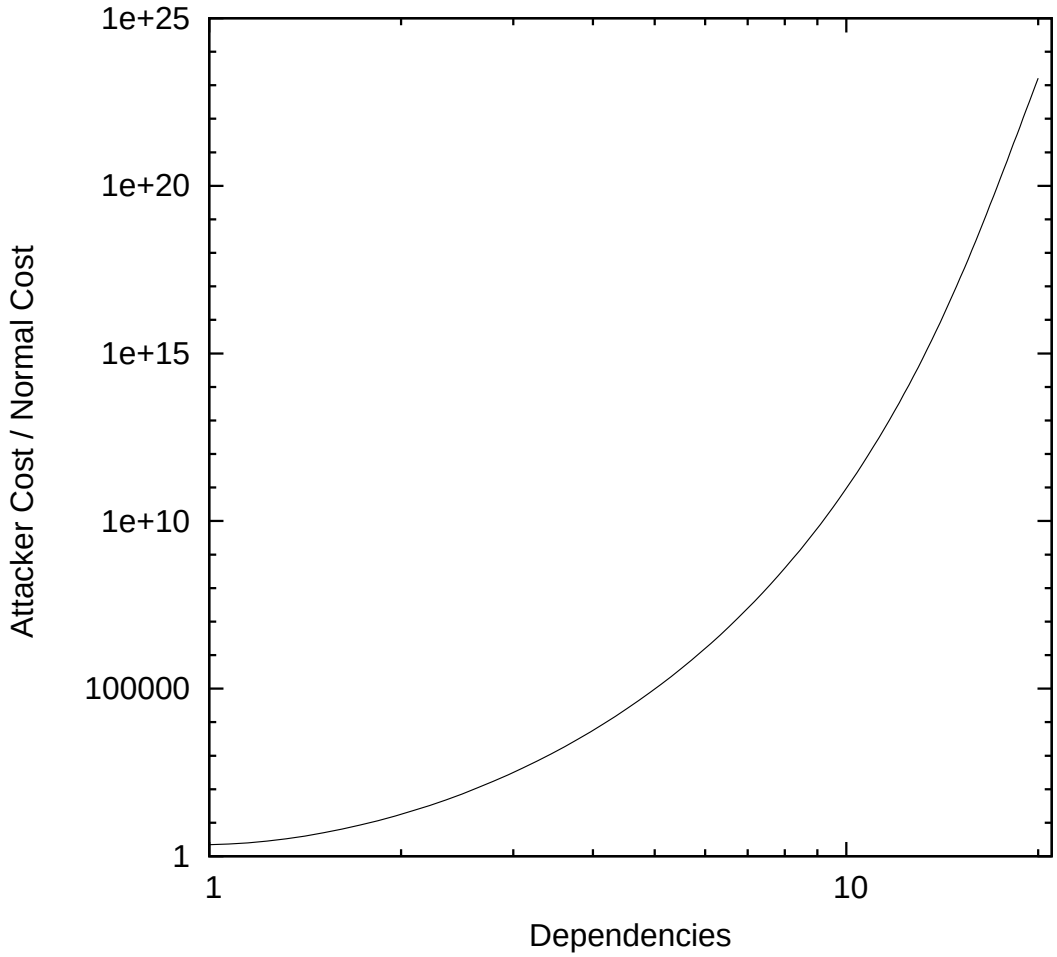

Figure 4.12: Time Space Attack Simulation Dependency

the pool size to 128 blocks. The attack tries to save 1 block. We set the number of dependencies at 4. We change the number of rounds from 1 to 20. Figure 4.13 shows the result of the simulation. The result shows that increasing the number of block computations drastically increases the cost of the time space trade off attack, however contrary to expectations the increasing number of block computations is better than increasing the number of dependencies for making the time space trade-off attack more expensive.

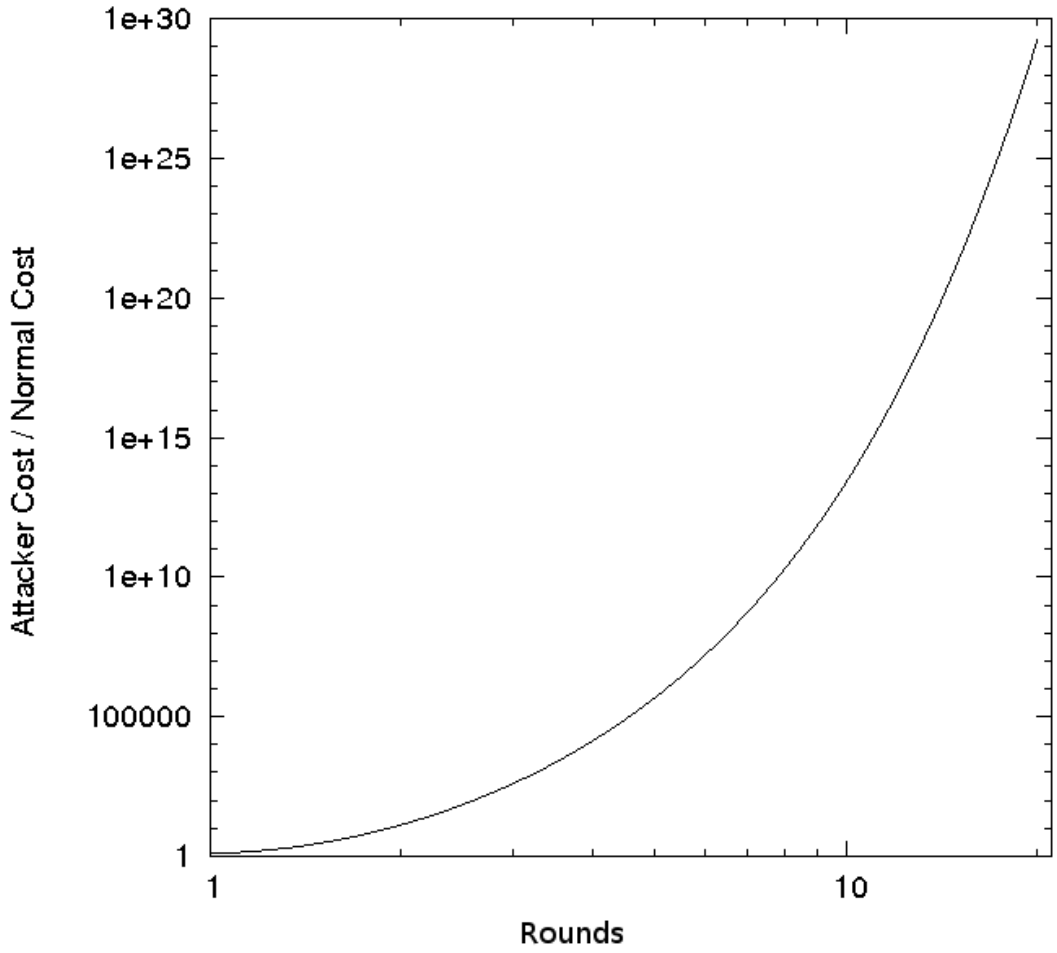

Figure 4.13: Time Space Attack Simulation Rounds

4.4 Experiments

We run some experiments to evaluate the performance of our update function. We performed these experiments on the ARM 1176 development board.

4.4.1 Pool Size Scalability

We describe the experiment to measure how the time it takes to compute the function changes as we change how much memory we use. We are measuring three different methods for performing the pool update, the CBC-like scheme, the pseudo-random scheme, and the public permutation scheme.

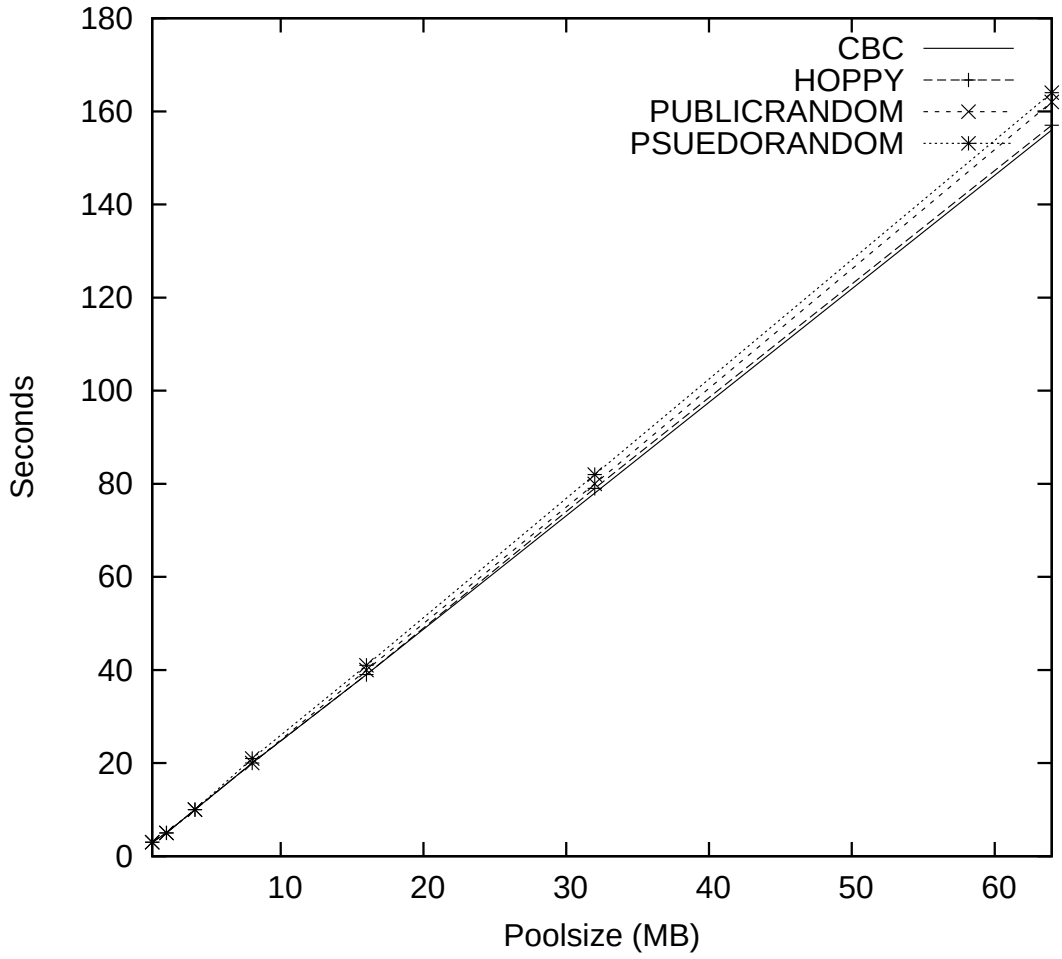

Figure 4.14: Pool Size Scalability

We fixed the number of rounds to compute the update at 1 and the number of dependencies at 2. We measure pool sizes ranging from 1 to 64 megabytes. We expect that the performance will not change too much across different block selection strategies, and we expect a linear increase in pool size to generate a linear increase in cost. Figure 4.14 shows the results of the experiment. The results match expectation.

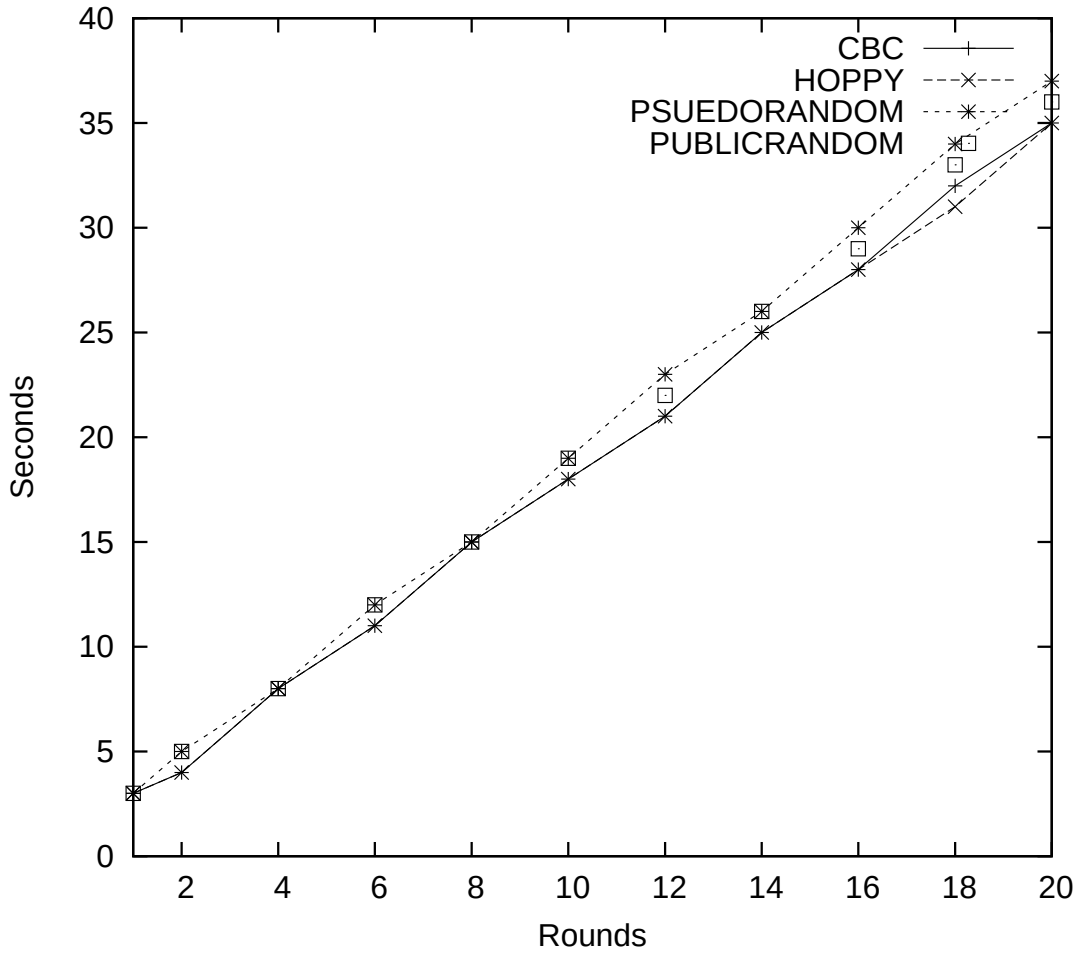

Figure 4.15: Round Scalability

4.4.2 Round Scalability

We describe the experiment to measure how the time it takes to compute the function changes as we change how many rounds we use to compute the function. We are measuring three different methods for preforming the pool update, the CBC-like scheme, the pseudo-random scheme, and public permutation scheme. We fixed the number of dependencies at 2 and the pool size at 1 megabyte. We vary the number of rounds from 1 to 20. We expect that the performance will not change too much across different block selection strategies, and we expect a linear increase in rounds to generate a linear increase in cost. Figure 4.15 shows the result of the experiment. The results match

expectation.

4.4.3 Dependency Scalability

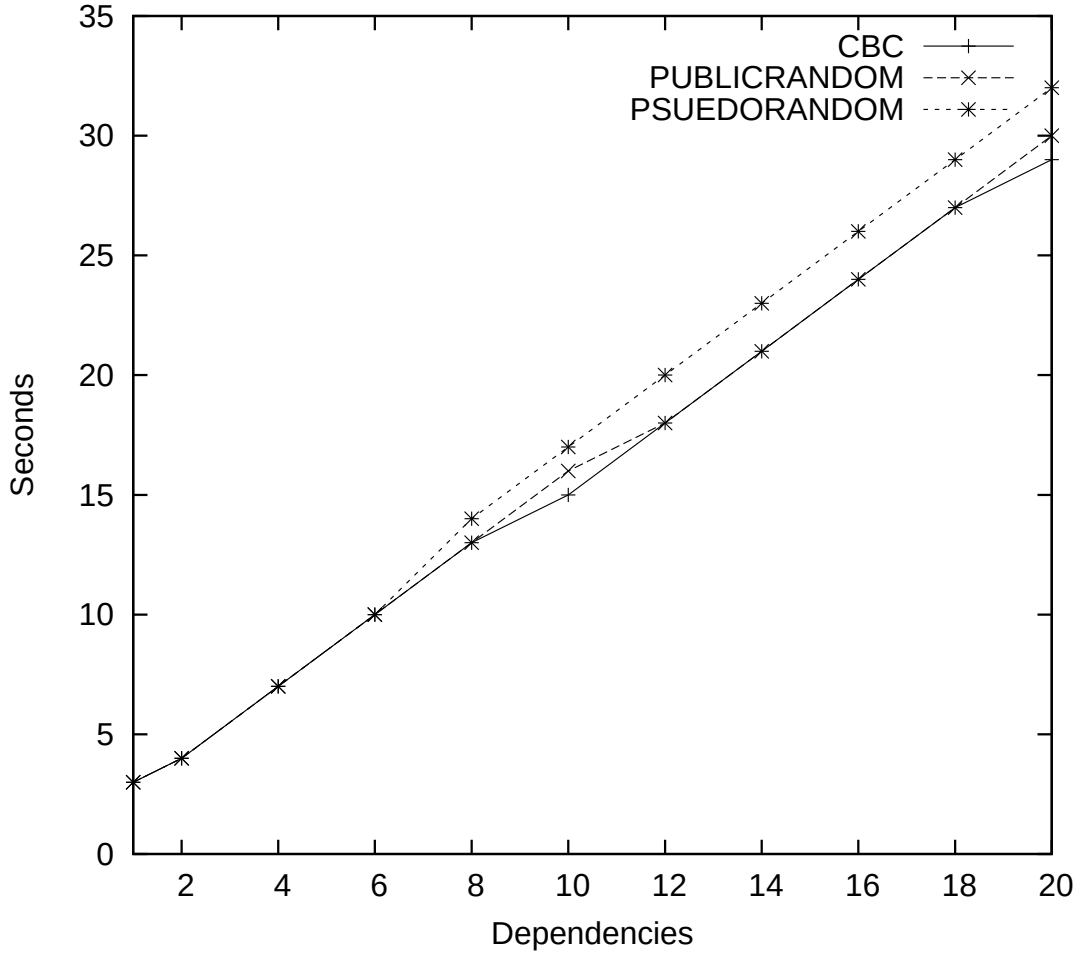

Figure 4.16: Dependency Scalability

We investigate our the performance of the pool update changes as we changes the number of dependencies in the computation of the pool update. We are measuring three different methods for preforming the pool update, the CBC-like scheme, the pseudo-random scheme, and public permutation scheme. We fixed the pool size at 64 megabytes and the number of rounds at 1. We vary the number of dependencies from 1 to 20. We expect that the performance will not change too

much across different block selection strategies, and we expect a linear increase in rounds to generate a linear increase in cost. Figure 4.16 shows the result of the experiment. The results match expectation.

Overall the results show that the pool update is efficient to compute.

Chapter 5

Conclusions

Cost pressure on the development, deployment and operation of large-scale embedded systems will continue to lead to the use of low-cost devices, which lack specialized security hardware, and often include low-assurance software and insecure management tools, for the foreseeable future. This suggests that (1) malware infestation of remote, embedded devices will continue to be a significant threat, and (2) secure operation and management of these devices has to be achieved despite this threat – a significant challenge for any system today. The mechanisms proposed in this paper address this challenge for remote-device authentication and establishment of malware-free device states. Our mechanisms remove the dependencies on precise timing and micro-architectural details (e.g., instruction timings) that have plagued traditional software-based attestation mechanisms, and form a sound basis for developing other robust primitives (e.g., cryptographic primitives resilient to side-channel attacks) for secure application development.

Chapter 6

Acknowlegements

This research was supported in part by Cylab at Carnegie Mellon under grants DAA19-02-1-0389 from the Army Research Office, and by grant NGIT2009100109 from the Northrop Grumman Information Technology Inc Cybersecurity Consortium. The views and conclusions contained here are those of the authors and should not be interpeted as necessarily representing the offical policies or endorsements, either express or implied, of ARO, CMU, CyLab, NGC, or the U.S.-Government or any of its agencies.

Bibliography

[1] Joel Alwen, Yevgeniy Dodis, Moni Naor, Gil Segev, Shabsi Walfish, and Daniel Wichs. Public-key encryption in the bounded-retrival model. 2009.

[2] John Alwen, Yevgeniy Dodis, and Danuiel Wichs. Survery: Leakage-resilience and the bounded retrieval model.

[3] ARM. ARM security technology. PRD29-GENC-009492C, 2009.

[4] Zvika Brakerski, Yael Tauman Kalai, Jonathan Katz, and Vinod Vaikuntanathan. Cryptography resilient to continual memory leakage. Cryptology ePrint Archive, Report 2010/278, 2010. `http://eprint.iacr.org/`.

[5] David Cash, Yan Zhong Ding, Yevgeniy Dodis, Wenke Lee, Richard Lipton, and Shabsi Walfish. Intrusion-resilient key exchange in the bounded retrieval model. In *TCC*, 2007.

[6] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, November 2009.

[7] Y.-G. Choi, J. Kang, and D. Nyang. Proactive code verification protocol in wireless sensor network. In *Proceedings of ICCSA*, 2007.

[8] Giovanni Di Crescenzo, Richard J. Lipton, and Shabsi Walfish. Perfectly secure password protocols in the bounded retrieval model. In *TCC*, pages 225–244, 2006.

[9] Mike Davis. SmartGrid device security: Adventures in a new medium. Black Hat USA, 2009.

[10] Yevgeniy Dodis, Kristiyan Haralambiev, Adriana Lopez-Alt, and Daniel Wichs. Cryptography against continuous memory attacks. Cryptology ePrint Archive, Report 2010/196, 2010. `http://eprint.iacr.org/`.

[11] Yevgeniy Dodis and Krzysztof Pietrzak. Leakage-resilient pseudorandom functions and side-channel attacks on feistel networks. In *CRYPTO*, pages 21–40, 2010.

[12] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, 2001.

[13] Stefan Dziembowski. Intrusion-resilience via the bounded-storage model. In *TCC*, pages 207–224, 2006.

[14] Stefan Dziembowski and Kryzsztof Pietrzak. Leakage-resilient cryptography. In *FOCS*, 2008.

[15] Sebastian Faust, Eike Kiltz, Krzysztof Pietrzak, and Guy N. Rothblum. Leakage-resilient signatures. In *TCC*, pages 343–360, 2010.

[16] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of ACM Conference on Computer and Communication Security (CCS)*, pages 148–160, 2002.

[17] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. *Lecture Notes in Computer Science*, 1999.

[18] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. *Lecture Notes in Computer Science*, 1996.

[19] Silvio Micali and Leonid Reyzin. Physically observable cryptography.

[20] Ravikanth Pappu. *Physical One-Way Functions*. PhD thesis, MIT School of Architecture and Planning, Program in Media Arts and Sciences, March 2001.

[21] Daniele Perito and Gene Tsudik. Secure code update for embedded devices via proofs of secure erasure. In *Proceedings of ESORICS*, 2010.

[22] Krzysztof Pietrzak. A leakage-resilient mode of operation. In *EUROCRYPT*, pages 462–482, 2009.

[23] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. Using fire and ice for detecting and recovering compromised nodes in sensor networks. Technical Report CMU-CS-04-187, School of Computer Science, Carnegie Mellon University, December 2004.

[24] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SCUBA: Secure code update by attestation in sensor networks. In *Proceedings of ACM Workshop on Wireless Security (WiSe)*, September 2006.

[25] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWAtt: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[26] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWATT: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[27] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim. Remote software-based attestation for wireless sensors. In *Proceedings of ESAS*, 2005.

[28] Trusted Computing Group. Trusted platform module main specification. Version 1.2, Revision 94, 2006.

[29] Trusted Computing Group. Trusted platform module main specification, Part 1: Design principles, Part 2: TPM structures, Part 3: Commands. Version 1.2, Revision 103, July 2007.

[30] Y. Yang, X. Wang, S. Zhu, and G. Cao. Distributed software-based attestation for node compromise detection in sensor networks. In *Proceedings of IEEE SRDS*, 2007.