6.857
NETWORK AND COMPUTER SECURITY
PROBLEM SET 1, PROBLEM 3

JOHN WANG

1. Problem 3.a

The Keccak function (selected to be SHA-3 by the NIST competition) had a very well defined set of design goals. In particular, since the NIST defined the requirements of SHA-3 explicitly, many of the goals that Keccak attains are set specifically for SHA-3. The NIST wanted a cryptographic hash function that would replace SHA-2 and use an entirely different algorithm. They set goals broadly in the following categories:

- Security. The algorithm should have a high expected safety margin and be resistant to known attacks, such as differential crypanalysis.
- Speed. The algorithm should have fast computation time. An algorithm that is highly secure but is infeasible in practice is useless for a cryptographic standard. The NIST asked for fast runtimes for both software and hardware implementions of the algorithm.
- Ease of use. The algorithm should not have too many parameters to tune and should also be easily implemented in either software or hardware. Widespread use of a cryptographic algorithm, such as SHA-3, would require it to be simple enough to be distributed to non-experts.
- Memory efficiency. The algorithm should not require an inordinately large amount of memory in order to run. Such an algorithm, even if effective, might hinder some users with older hardware or memory constraints.

These goals influenced a number of design decisions made by the Keccak team. For instance, simplicity and functionality greatly affect the choice of the sponge construction for Keccak. The sponge construction's simplicity meant that its security bounds could be expressed in a simple way, and also, that those wanting to implement Keccak would have an easier time understanding the algorithm. The iterated permutation of Keccak was chosen in part because of its memory efficiency. Instead of using a feed-forward loop, like in many other cryptographic algorithms, Keccak's iterated permutation only required the state of the last iteration, which meant that Keccak only required a small amount of memory.

Another goal of SHA-3 was ease of use, and this led to a number of design changes by the Keccak team. For instance, the Keccak function is actually a variable-input variable-output function. This means that the user has the ability to choose the length of the output from the Keccak function. The Keccak team, however, made a deliberate decision to choose particular input-output length pairs. The reasoning behind this decision stemmed from the fact that the input length b is composed of two parameters r and c such that $b = r + c$. The capacity c governs the tradeoff between speed and security. Larger values of c cause the function to be more secure, but also decrease the function's performance. The creators of Keccak said, "the choice of the capacity value determines a ceiling to the security level that the sponge function provides and one could argue that the user usually does not have the responsibility or the expertise to make that choice." Hence, the team not only defined predetermined pairs of input and output lengths, but they also created a rule of thumb, telling the users to set the capacity as $c = 2n$ for an output message of length n.

One other consequence of these particular design goals was a method for responding to a breach in Keccak's security. Rivest originally suggested that having a tunable security parameter, i.e. increasing the number of rounds of the permutation, could add extra security in event of a breach. However, the Keccak team decided against this suggestion because of extra burden on the user and the additional implementation complexity. Instead, they chose to use the same algorithm and insert a number of waste bits so that the actual bitrate r would become $r - \delta$, essentially increasing the capacity by some δ. Again, this solution was made in the view of user friendliness, in the expectation of widespread use of SHA-3.

The Keccak team made a number of assumptions about the types of inputs Keccak would receive, and also, about the properties of the permutation functions in Keccak. First, the Keccak team assumed that the asymmetry in the ι permutation was sufficient to produce enough asymmetry to prevent attacks. Since the

Keccak function is symmetric by design (for easy of implementation), if the ι function is not as asymmetric as previously believed, then Keccak could be vulnerable to slide attacks between rounds.

Second, the Keccak team assumed that it would remain difficult for an adversary to construct higher order differentials for the Keccak-f function. The Keccak team believes that the high average diffusion of the Keccak function makes it impossible to create a practical differential.

Finally, the Keccak team assumed that it would remain difficult to solve a large system of simulatenous equations. Attacking Keccak can be reduced to solving a system of equations of $b(n_r + 1)$ variables and $b(n_r + 1) + n - r$ equations (where n_r is the total number of rounds). Thus, Keccak could be broken if new breakthroughs in equation solving occur.

An implicit assumption that the Keccak team makes is that the number of rounds will be sufficiently large, and will be equal to at least the number of suggested rounds. If the number of rounds decreases below the suggested number, then Keccak is vulnerable to a number of attacks. For instance, the number of equations and variables needed to solve the CICO (constrained input constrained output) problems grows linearly in Keccak with the number of rounds. Thus, a low number of rounds makes Keccak vulernable to attacks which attempt to solve these systems of equations. Other attacks are also more potent at low round numbers, especially because Keccak has better diffusion properties with higher numbers of rounds.

The */dev/random* function for producing random numbers has somewhat different design goals. The */dev/random* function was created and meant to be used as a random number generator on linux machines. In particular, the authors of the function meant for it to be strong and more secure than a pseudorandom number generator. The general idea is to use environmental noise and other hard to predict inputs to generate a pot of entropy. The entropy would then be stirred by a cryptographic hash.

Since the */dev/random* function was intended to be strong and secure, it has been used in many crucial applications such as the generation of SSL and RSA keys. The overarching goals of the function are as follows:

- Generate numbers which are both random and hard to predict by an attacker. If an attacker could predict the numbers generated by this function, it would compromise the ability for this function to create secret keys or pads, which was originally a use case.
- The generation of randomness should not significantly slow down the operating system. Since entropy could be added to the pool potentially on every interrupt, it is crucial that the method of adding entropy be fast.
- Fast recovery from compromise of the entropy pool. If an adversary obtains knowledge about the entropy pool, the function should be resilient. In addition, if the cryptographic hash function used is broken, the function should have countermeasures that prevent the random numbers it generates from being completely predictable.

However, the function makes many assumptions about the environment it operates in and the inputs it obtains. Some of these assumptions are more tenuous than others and have lead to weaknesses in */dev/random*. Some of the more critical assumptions are given below:

- The input values are unpredictable. The function assumes that the environmental input values cannot be copied or known by an adversary. For keyboard and mouse input by users, the assumption may hold. However, for servers with no human interaction, the environmental input comes from disk writes, time, and other somewhat predictable inputs which weakens the security of */dev/random*. This assumption is crucial especially during boot, when the input values are usually very predictable and can lead to vulnerabilities in the function.
- The SHA hash function is secure. Since the */dev/random* function uses SHA to mix the entropy pool and return random bits (by digesting the entire pool), the security of the */dev/random* function relies on the difficulty of predicting outputs from the SHA hash.
- The internal state of the entropy pool cannot be determined by an attacker. If the attacker could obtain access to the contents of the entropy pool, then the attacker could simulate the SHA digest and know the random numbers to be generated from */dev/random*.
- The entropy pool at system startup is unpredictable. As an added security feature, */dev/random* makes sure that the entropy pool persists between shutdown and startup. Since the boot process is predictable, the helps ensure that an attacker cannot predict random numbers while the computer is still booting. This assumption could break down, however, in cases where the computer has been shutdown for a long time and the entropy pool's bits are zeroed.

The assumptions underlying */dev/random* make it more easy to exploit, especially if an attacker knows about the interrupt operations on the machine in question and is able to attack the machine on startup. For most cases, however, */dev/random* generates keys which are hard to predict.

2. Problem 3.b

The proposal to use the sponge construction in */dev/random* would require a number of properties to be true about SHA-3 and the sponge construction it uses. Ben Bitdiddle's proposal means that every r new bits of information will be absorbed into the entropy pool by xor-ing them with the current state and calling the Keccak-f function on the new state. When r bits are extracted, then the Keccak-f function will be applied to the entropy pool.

This proposal, at face value, seems innocuous. However, taking a closer look at this proposal, one can see that the creator's of Keccak did not have this as an intended use. In particular, there are a couple of cases where the proposal would have weaknesses. The Keccak team assumed that the number of rounds of application of Keccak-f would be greater than or equal to the suggested number of rounds. The security and diffusion properties do not hold up well when few rounds are applied.

To illustrate a weakness in Ben Bitdiddle's proposal, imagine that a machine boots up and that an adversary knows the exact boot sequence of the machine. In this case, the adversary will know the beginning state of the entropy pool. Now, suppose the user has some keyboard or mouse input, and */dev/random* is called to generate some SSL keys. In this case, there is only a single round of Keccak-f application before the output from the randomization function is provided. It is known, however, that low numbers of rounds of Keccak-f are vulnerable to attack.

In particular, one could solve the CICO (constrained input constrained output) problem. It is well-known that Keccak can be solved with a system of equations of $(n_r + 1)b$ variables and $(n_r + 1)b + n_r - r$ equations, where n_r is the number of rounds. In fact, KeccakTools was written by the authors of Keccak to provide this system of equations. Since $n_r = 1$ in this case, the amount of computation required to solve this system is greatly reduced compared to a full application of Keccak.

There are other weaknesses that result from a low number of rounds. Amplified boomerang and rectangle attacks, which construct chains of differentials over small numbers of rounds, are more easily used for Keccak with a small number of rounds. Higher order differentials, the authors claim, do not pose a significant threat because of the high average diffusion of Keccak. However, the diffusion in Keccak occurs in proportion to the number of rounds. Therefore, with a small number of rounds, the average diffusion of Keccak may be low enough to make it vulnerable to higher order differential attacks.

Other attacks, such as impossible differentials, also work better with a lower number of rounds. The authors have provided empirical evidence that Keccak-f permutations behave randomly. The authors dismissed these attacks because they suggested that the number of rounds of Keccak was sufficient to avoid low-weight differential trails.

Therefore, most attacks are more potent when Keccak has a low number of rounds. The fact that Ben Bitdiddle's proposal for */dev/random* can be attacked on these grounds significantly weakens his proposal. For his proposal to work, the Keccak-f permutations would have to have a high security margin even for a low number of rounds.

The main reason Ben Bitdiddle's proposal has weaknesses is because after every r bits of environmental input, it is plausible that the user could retrieve r bits of output. Such fast turnaround prevents SHA-3 from being able to go through the requisite rounds of absorption and interleave the Keccak-f permutations with new bits. It is possible for only a single round of Keccak-f permutations to occur before output is obtained by the user. The authors never tried to prove many of the security properties of Keccak for such a low number of rounds.