**Identifying Unknown Malware Intrusion by Supercomputing**

**[Malware Detection & Identification by Tamper-proof Hardware]**

Our project is about detecting and exactly identifying malware injected in the computer machine. Specifically, we detect malware intrusion by using a special security hardware chip called TPM, and we quickly analyze the identity of malware by using OpenMP and CUDA.

**[Limitation of All Security Software]**

First we start with explaining why we ever need a security hardware chip on our machine. Actually, we all have some kind of security software installed on our laptop that finds and deletes malware and viruses. But such security software is often imperfect, because once some invents a very good security solution, there comes out some smarter attacks that can break it. And even if we fix the vulnerability and make our security stronger, there comes another attack to get around it. So cyber security and cyber attacks are in endless arms race. This is actually a big problem when it comes to Internet of Things, such as smart cars, because even a single security breach can kill the driver. So our fundamental question is, can we come up with a powerful security solution which no software attacks existing now and in the future can possibly break? The most promising answer is tamper-proof security hardware, such as TPM.

**[TPM Chip’s Security Features]**

TPM is a security chip installed on the computer’s motherboard, and this chip is independent form the CPU chip. While the computer is turned on, the CPU can communicate with the TPM to use hardware-guaranteed security services, such as digital signing and storing a summary of the computer system’s state. TPM internally has its own registers and its RSA private key is burned into its read-only memory at its manufacture stage, and this private key never leaves the TPM. If someone physically opens the TPM to peep inside, its private key gets evaporated and lost forever. These days, the TPM is mainly used for two purposes: system integrity measurement and remote attestation, which we’ll cover.

**[Step 1: TPM-enabled System Booting (1) ]**

First we’ll explain TPM’s system integrity measurement. This figure is an example of a system boot-up software stack, which is a history of executed programs on the computer machine since its power has been turned on. In this example, the first program executed on the computer was BIOS, then the GRUB2 bootloader gets executed, and then Linux kernel image, and then various user-space startup programs, including kernel modules and dynamically shared libraries. Whenever these programs are loaded on the memory, their hash values are computed and overwritten to the TPM’s special register called PCR. This equation on the slide describes how the PCR is updated. The TPM concatenates the current PCR’s value with the hash value of the newly loaded program, and computes a new hash value by using SHA1 digest algorithm and stores it in the PCR. And this PCR update computation is done whenever a new program is loaded and executed on the memory. So the PCR value at any point in time represents the summary of all programs being executed so far since the power has been turned on. And this summary value is unique, because SHA1 algorithm’s input/output space is very large and it is collision-free. So, once our computer finishes booting up, we can check the TPM’s current PCR value to verify if our machine’s software stack is in the expected secure state.

**[Step 1: TPM-enabled System Booting (2) ]**

But suppose there was some malware injected into our machine and it got executed in the middle of the software stack. Then when this malware gets launched, this malware’s hash value will be combined into the TPM’s PCR, so the finally resulting PCR value will become completely different from what we expect. By checking this PCR value, we know that our system is not in the expected state.

**[Malware Identification]**

Our project’s goal is to reverse-engineer and find out the exact identity of this injected malware. Basically, we insert each malware candidate in each possible position in the verified system boot-up sequence, and simulate the computation of the PCR value, and check if the simulated final PCR value coincides with the TPM’s actual PCR value. If yes, the candidate malware we just chose is the injected malware. But as the number of malware being injected increases, the search space for malware reverse-engineering increases dramatically and the computation becomes very heavy. So, to solve this problem more quickly and efficiently, we will use parallel computing tools such as OpenMP and CUDA, and demonstrate if they are practical enough.

**[Remote Attestation]**

Remote attestation is a technique to remotely analyze the target computer’s PCR in a secure way. Basically, we send a remote message to the target computer to send its TPM’s PCR value. Then, the TPM signs its current PCR value with its private key and remotely sends it to us. Then we verify the signature on the PCR value to check if it was really signed by the TPM. If the signature is verified, we march into the malware identification stage, which is using OpenMP and CUDA.

**[Experiment: Remote Attestation]**

In the first experiment, we tested the delay of remote attestation on various network latencies and throughputs, and the delay was between 0.5 to 4 seconds, which is reasonable and acceptable.

**[Experiment: Malware Identification]**

In the second experiment, we tested the delay for malware identification. We used 200 binary files as the expected software stack, and injected 1 to 4 malware into this sequence. Our result shows that when there were 4 malware injected, the sequential computing takes more than 250 hours to identify those 4 malware, whereas OpenMP takes about 25 hours, and CUDA takes roughly 2.5 hours. So OpenMP achieves up to 10 times speedup, and CUDA achieves up to 100 times speedup.

**[Conclusion]**

We conclude that CUDA can quickly compute and identify up to 4 simultaneously injected malwares within 2-3 hours. Our future research is to efficiently identify self-modifying malware which randomly changes its binary representation. Another research possibility is to find a way to force a deterministic software stack for multi-core processes whose software launch sequence can locally vary.