One-Time Programs Made Practical

Lianying Zhao[1(✉)], Joseph I. Choi[2], Didem Demirag[3], Kevin R. B. Butler[2], Mohammad Mannan[3], Erman Ayday[4], and Jeremy Clark[3]

[1] University of Toronto, Toronto, ON, Canada
lianying.zhao@utoronto.ca
[2] University of Florida, Gainesville, FL, USA
[3] Concordia University, Montreal, QC, Canada
[4] Case Western Reserve University, Cleveland, OH, USA

Abstract. A one-time program (OTP) works as follows: Alice provides Bob with the implementation of some function. Bob can have the function evaluated exclusively on a single input of his choosing. Once executed, the program will fail to evaluate on any other input. State-of-the-art one-time programs have remained theoretical, requiring custom hardware that is cost-ineffective/unavailable, or confined to ad-hoc/unrealistic assumptions. To bridge this gap, we explore how the Trusted Execution Environment (TEE) of modern CPUs can realize the OTP functionality. Specifically, we build two flavours of such a system: in the first, the TEE directly enforces the one-timeness of the program; in the second, the program is represented with a garbled circuit and the TEE ensures Bob's input can only be wired into the circuit once, equivalent to a smaller cryptographic primitive called one-time memory. These have different performance profiles: the first is best when Alice's input is small and Bob's is large, and the second for the converse.

1 Introduction

Consider the well-studied scenario of secure two-party computation: Alice and Bob want to compute a function on their inputs, but they do not want to disclose these inputs to each other (beyond what can be inferred from the output of the computation). This is traditionally handled by an interactive protocol between Alice and Bob.[1] In this paper, we instead study a non-interactive protocol as follows: Alice prepares a device for Bob with the function and her input included; once Bob receives this device from Alice, he supplies his input and learns the outcome of the computation. The device will not reveal the outcome for any additional inputs (thus, a one-time program [12]). Alice might be a company selling the device in a retail store, and Bob the customer; the two never interact directly. By using the device offline, Bob is assured that his input remains private.

To build a one-time program (OTP), we use the Trusted Execution Environment (TEE), a hardware-assisted secure mode on modern processors, where

[1] Hazay and Lindell [19] give a thorough treatment of interactive two-party protocols.

© International Financial Cryptography Association 2019
I. Goldberg and T. Moore (Eds.): FC 2019, LNCS 11598, pp. 646–666, 2019.
https://doi.org/10.1007/978-3-030-32101-7_37

execution integrity and secrecy are ensured [31], with qualities that include platform state binding and protection of succinct secrets. TEEs may appear to offer a trivial solution to OTPs; however, complexities arise due to Bob's physical possession of the device and, more importantly, performance issues. We propose two configurations for one-time programs built on TEEs: (1) deployed directly in the TEE, and (2) deployed indirectly via TEE-backed one-time memory (OTM) [12] and garbled circuits [50] outside of the TEE. OTMs hold two keys, only one of which gets revealed (dependent on its input); the other is effectively destroyed.

Contributions. Our system, built using Intel Trusted Execution Technology (TXT) [13] and Trusted Platform Module (TPM) [45] as the TEE, is available today (as opposed to custom OTP/OTM implementations using FPGA [21], PUF [24], quantum mechanisms [5], or online services [25]) and could be built for less than $500.[2]

We propose and implement the following OTP variants, considering that TPM-sealing[3] or encrypting data is time-consuming.

- *TXT-only* seals/unseals Alice's input directly, and performance is thus sensitive to Alice's input size. Bob's input is entered in plaintext and processed in TXT after he has received the device.
- *GC-based* converts the logic into garbled circuit, where number of key pairs is determined by Bob's input size. Key pairs are encrypted/decrypted with a master key (MK). This way, the performance is largely determined by Bob's input size. Upon receiving the device, he does the one-time selection of key pairs in TXT to reflect his input. Thereafter, evaluation of the garbled circuit can be done on any machine with the selected keys.

To illustrate the generality of our solution, we also map the following application into our proposed OTP paradigm: a company selling devices that will perform a private genomic test on the customer's sequenced genome. For this use case, in one of our two variants (TXT-only), a company can initialize the device in 5.6 s and a customer can perform a test in 34 s.

2 Preliminaries

2.1 One-Time Program Background

A one-time program can be conceived of as a non-interactive version of a two party computation: $y = f(a, b)$ where a is Alice's private input, b is Bob's, f is a public function (or program), and y is the output. Alice hands to Bob an implementation of $f_a(\cdot)$ which Bob can evaluate on any input of his choosing: $y_b = f_a(b)$. Once he executes on b, he cannot compute $f_a(\cdot)$ again on a different input. For our practical use-case, we conceive of OTPs with less generality

[2] As an example, Intel STK2mv64CC, a Compute Stick that supports both TXT and TPM, was priced at $499.95 USD on Amazon.com (as of September 2018).

[3] A state-bound cryptographic operation performed by the TPM chip, like encryption.

as originally proposed by Goldwasser et al. [12]; essentially we treat them as one-time, non-interactive programs that hide Alice and Bob's private inputs from each other without any strong guarantees on f itself. Note with a general compiler for f (which we have for both flavours of our system), it is easy but inefficient to keep f private.[4]

2.2 Threat Model and Requirements

We informally consider an OTP to be secure if the following properties are achieved: (1) Alice's input a is confidential from Bob; (2) Bob's input b is confidential from Alice, and (3) no more than one b can be executed in $f(a, b)$ per device. We argue the security of our two systems in Sect. 8 but provide a synopsis here first. Property 3 is enforced through a trusted execution environment, either directly (TXT-only variant in Sect. 4) or indirectly via a one-time memory device (GC-based TXT in Sect. 5) as per the Goldwasser et al. construction. Given Property 3, we consider Property 1 to be satisfied if an adversary learns at most negligible information about a when they choose b and observe $\langle \mathsf{OTP}, f(a, b), b\rangle$ as opposed to simply $\langle f(a, b), b\rangle$, where OTP is the entire instantiation of the system, including the TPM-sealed memory and system details (and for the GC-variant: the garbled circuit and keys revealed through specifying b). Property 2 is achieved by being provisioned an offline device that can compute $f_a(b)$ without any interaction with Alice. There is a possibility that the device surreptitiously stores Bob's input and tries to leak it back to Alice. We discuss this systems-level attack in Sect. 8. We also address a subtle adaptive security attack in the full version of our paper.

The selection of TEE has to reflect the aforementioned Properties 1 and 3. Property 3 is achieved by stateful (recording the one-time state) and integrity-protected (enforcing one-timeness) execution, which is the fundamental purpose of all today's TEEs. Moreover, both Properties 1 and 3 mandate no information leakage, which can occur through either software or physical side-channels. We choose Intel TXT, primarily because of its *exclusiveness*, which means: TXT occupies the entire system when secure execution is started and no other code can run in parallel. This naturally avoids all software side-channels, an advantage over non-exclusive TEEs. We do consider using non-exclusive TEEs as future exploration when the challenge of software side-channels has been overcome, e.g., for Intel SGX, the (recent) continually identified side-channel attacks, such as Foreshadow [6], branch shadowing [29], cache attacks [4], and more; for ARM TrustZone, there have been TruSpy [51], Cachegrab [36], etc. They all point to the situation when trusted and untrusted code run on shared hardware.

The known physical side-channels can also be mitigated in the setting of our OTP, i.e., DMA attacks are impossible if I/O protection is enable

[4] Essentially, one would define a very general function we might call Apply that will execute the first input variable on the second: $y = \mathsf{Apply}(f, b) = f(b)$. Since f is now Alice's private input, it is hidden. The implementation of Apply might be a universal circuit where f defines the gates' logic—in this case Apply would leak (an upper-bound on) the circuit size of f but otherwise keep f private.

(by the chipset), and the cold-boot attack [17] can be avoided if we choose computers with RAM soldered on the motherboard (cannot be removed to be mounted on another machine, see Sect. 8).

We strive for a reasonable, real-world threat model where we mitigate attacks introduced by our system but do not necessarily resolve attacks that apply broadly to practical security systems. Specifically, we assume:

- Alice is monetarily driven or at least curious to learn Bob's input, while Bob is similarly curious to learn the algorithm of the circuit and/or re-evaluate it on multiple inputs of his choice.
- We assume Alice produces a device that can be reasonably assured to execute as promised (disclosed source, attestation quotes over an integral channel, and no network capabilities).
- We assume that Alice's circuit (including the function and her input) actually constitutes the promised functionality (e.g., is a legitimate genomic test).
- We assume the sound delivery of the device to Bob. We do not consider devices potentially subverted in transit which applies to all electronics [40].
- Both Alice and Bob have to trust the hardware manufacturer (in our case, Intel and the TPM vendor) for their own purposes. Alice trusts that the circuit can only be evaluated once on a given input from Bob, while Bob trusts that the received circuit is genuine and the output results are trustworthy.
- Bob has only bounded computational power, and may go to some lab effort, such as tapping pins on the motherboard and cloning a hard drive, but not efforts as complicated as imaging a chip [27,28,43].
- Components on the motherboard cannot be manipulated easily (e.g., forwarding TPM traffic from a forged chip to a genuine one by desoldering).

2.3 Intel TXT and TPM

Intel Trusted Execution Technology (TXT) is also known as "late launch", for its capability to launch secure execution at any point, occupying the entire system. When the CPU enters the special mode of TXT, all current machine state is discarded/suspended and a fresh secure session is started, hence its exclusiveness, as opposed to sharing hardware with untrusted code.

Components. TXT relies on three mandatory hardware components to function: (a) CPU. The instruction set is extended with a few new instructions for the management of TXT execution. (b) Chipset. The chipset (on the motherboard) is responsible for enforcing I/O protection such that the specified range of I/O space is only accessible by the protected code in TXT; and (c) TPM. Trusted Platform Module [45] is a microchip, serving as the secure storage (termed Secure Element). Its *PCR* (Platform Configuration Register) is volatile storage containing the machine state, in the form of concatenated hash values. There are also multiple PCRs for different purposes. On the TPM, there is also non-volatile storage (termed *NVRAM*), allocated in the unit of *index* of various sizes. Multiple indices can be defined depending on the capacity of a specific TPM model.

Measured Launch. A provisioning stage is always involved where the platform is assumed trusted and uncompromised. A piece of code is measured (similar to hashing) and the measurements are stored in certain TPM NVRAM indices as policies. Thereafter (in our case in the normal execution mode with Bob), the program being loaded is measured and compared with the policies stored in TPM. The system may then abort execution if mismatch is detected, or otherwise proceed. This process is enforced by the CPU.

Machine State Binding. As run-time secrecy (secret in use) is ensured by measured launch and I/O isolation, we also need secrecy for stored data (secret at rest). Alice's input should not be learned by Bob when the device is shipped to him. From the start of TXT execution, each stage measures the next stage's code and *extends* the hash values as measurement to the PCR (concatenated and hashed with the existing value). This way, the measurements are chained, and at a specific time the PCR value reflects what has been loaded before. The root of this chained trust is the measured launch.

Such chained measurements (in PCRs) can be used to derive the key for data encryption, so that only when a desired software stack is running can the protected data be decrypted. This cryptographic operation performed by the TPM is termed *sealing*. A piece of data sealed under certain PCRs can only be unsealed under the same PCRs, hence bound to a specific machine state. The sealed data (ciphertext) can be stored anywhere depending on its size. It is noteworthy to mention that there exists a distinct equivalent of sealing which, instead of just encryption, stores data in a TPM NVRAM index and binds its access to a set of PCRs. As a result, without the correct machine state, the NVRAM index is completely inaccessible (read/write) and thus replaying the ciphertext is prevented. We term it *PCR-bound NVRAM sealing* in this paper and use it for our OTP prototype implementation.

3 Related Work

In the original one-time program paper by Goldwasser et al. [12], OTM is left as a theoretical device. In the ensuing years, there have been some design suggestions based on quantum mechanisms [5], physically unclonable functions [24], and FPGA circuits [21]. (a) Järvinen et al. [21] provide an FPGA-based implementation for GC/OTP, with a GC evaluation of AES, as an example of a complex OTP application. They conclude that although GC/OTP can be realized, their solution should be used only for "truly security-critical applications" due to high deployment and operational costs. They also provide a cryptographic mechanism for protecting against a certain adaptive attack with one-time programs; it is tailored for situations where the function's output size is larger than the length of a special holdoff string stored at each OTM. (b) Kitamura et al. [25] realize OTP without OTM by proposing a distributed protocol, based on secret sharing, between non-colluding entities to realize the 'select one key; delete the other key' functionality. This introduces further interaction and entities. Our approach is in the opposite direction: removing all interaction (other

than transfer of the device) from the protocol. (c) Prior to OTP being proposed, Gunupudi and Tate [16] proposed count-limited private key usage for realizing non-interactive oblivious transfer using a TPM. Their solution requires changes in the TPM design (due to lack of a TEE). In contrast, we utilize unmodified TPM 1.2. (d) In a more generalized setting, ICE [41] and Ariadne [42] consider the state continuity of any stateful program (including N-timeness) in the face of unexpected interruption, and propose mechanisms to ensure both rollback protection and usability (i.e., liveness). We solve the specific problem of one-timeness/N-timeness, focusing more on how to deal with input/output and its implication on performance. We do sacrifice liveness (i.e., we flip the one-timeness flag upon entry and thus the program might run zero time if crashed halfway). We believe their approaches can be applied in conjunction with ours.

4 System 1: TXT-Only

Overview. In the first system, we propose to achieve one-timeness by running the protected program in TEE only once (relying on logic integrity) and storing its persistent state (e.g., the one-time indicator) in a way that it is only accessible from within the TEE. To eliminate information leakage from software side-channels, we have chosen Intel TXT for its exclusiveness (i.e., no other software in parallel).[5] We hence name this design *TXT-only*.

To achieve minimal TCB (Trusted Computing Base) and simplicity, we choose native C programming in TXT (as opposed to running an OS/VM). Therefore, for one-time programs that have an existing implementation in other languages, per-application adaptation is required (cf. similar porting effort is needed for the GC-based variant in Sect. 5). New programs may not require extra effort.

Design. We briefly describe the components and workflow of the TXT-only system as follows. A one-time indicator (flag) is sealed into the PCR-bound TPM NVRAM to prevent replay attacks. The indicator is checked and then flipped upon entry of the OTP. Without network connection, the device shipped to the client can no longer leak any of the client's secrets to the vendor. Therefore, only the vendor's secret input has to be protected. We TPM-seal the vendor input on hard drive for better scalability, and there is no need to address replay attacks for vendor input as one-timeness is already enforced with the flag.

The OTP program is loaded by the Intel official project *tboot* [20] and GRUB. It complies with the Multiboot specification [11], and for accessing TPM, we reuse part of the code from tboot, and develop our own functions for commands that are unavailable elsewhere, e.g., reading/writing indices with PCR-bound NVRAM sealing. Since we do not load a whole OS into TXT with tboot, we cannot use OS services for disk I/O access; instead, we implement raw PATA (Parallel ATA, a legacy interface to the hard drive, compatible mode with SATA) logic and directly access disk sectors with DMA (Direct Memory Access).

[5] We consider various TEEs and justify this choice in the full version of our paper.

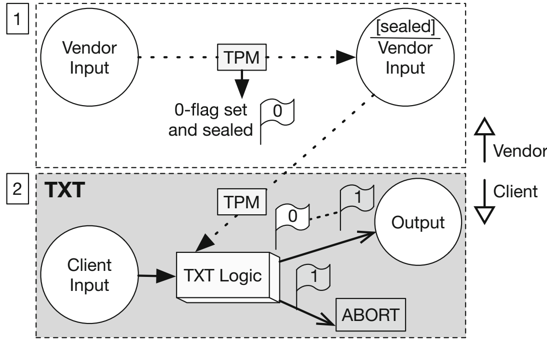

Fig. 1. Our realization of OTPs spans two phases when relying on TXT alone for the entire computation. Alice is active only during phase 1; Bob only during phase 2.

In the *provisioning mode*, the OTP program performs a one-time setup, such as initiating the flag in NVRAM, sealing (overwriting) Alice's secret, etc. Once the *normal execution mode* is entered, the program will refuse to run a second time.

Memory Exposure. As an optional feature for certain computers with swappable RAM, we expose the unsealed vendor input in very small chunks during execution. For example, if the vendor input has 100 records, we would unseal one record into RAM each iteration for processing the whole user input. This way, in case of the destructive cold boot attack, the adversary only learns one-hundredth of the vendor's secret, and no more attempts are possible (the indicator is already updated).

4.1 TXT-Only Provisioning/Evaluation.

Figure 1 gives an overview of *TXT-only*, illustrating the initial provisioning by Alice and evaluation of the function upon delivery to Bob. Note that what is delivered to Bob is the entire computer in our prototype (laptop or barebone like Intel NUC).

Provisioning at Alice's Site. At first, Alice is tasked with setting up the box, which will be delivered to Bob. Alice performs the following: (1) Write the integrity-protected payload/logic in C adapted to the native TXT environment, e.g., static-linking any external libraries and reading input data in small chunks. We may refer to it as the TXT program thereinafter. (2) In the provisioning mode, initialize the flag to 0 and seal.[6] The one-timeness flag is stored with the PCR-bound NVRAM sealing. Instead of depending on a password and regular sealing, this is like stronger access-controlled ciphertext. (3) Seal Alice's input onto the hard drive.

[6] A flag is more straightforward to implement than a TPM monotonic counter, thanks to the PCR-bound NVRAM sealing, whereas a counter would involve extra steps (such as attesting to the counterAuth password).

Evaluation at Bob's Site. After receiving the computation box from Alice, Bob performs the following: (1) Place the file with Bob's input on the hard drive. (2) Load the TXT program in normal execution mode, which will read in Bob's input and unseal Alice's input to compute on. (3) Receive the evaluation result (e.g., from the screen or hard drive). As long as it is Bob's first attempt to run the TXT program, the computation will be permitted and the result will be returned to Bob. Otherwise, the TXT program will abort upon loading in step (2), as shown in Fig. 1.

5 System 2: GC-Based

As seen in our TXT-only approach to OTP (System 1) the data processing for protection is only applied to Alice's input (with either sealing/unsealing or encryption/decryption), and Bob's input is always exposed in plaintext due to the machine's physical possession by Bob. Intuitively, we may think that it is a good choice when Alice's input is relatively small regardless of Bob's input size. However, there might be other applications where Alice's input is substantially larger and become the performance bottleneck. Is there a construction that complements TXT-only and is less sensitive to Alice's input size? The answer may lie in garbled circuits. During garbled circuit execution, randomly generated strings (or keys) are used to iteratively unlock each gate until arriving at the final output. Alice's input (size) is only "reflected" in the garbled circuit (assumed not trivially invertible [12]), and the key pairs (whose number is determined by Bob's input size, not to do with Alice's) are sealed/encrypted, hence insensitive to Alice's input size.

To adapt garbled circuits for OTP, key generation and key selection steps are separated. As long as we limit key selection to occur a single time, and the unchosen key of each key pair is never revealed, we can prevent running a particular circuit on a different input. To prevent keys from being selected more than once, we need to instantiate a one-time memory (OTM), which reveals the key corresponding to each input bit and effectively destroys (or its equivalent) the unchosen key in the key pair. OTM is left as a theoretical device in the original OTP paper [12]. We realize it using Intel TXT and the TPM. As in System 1, we seal a one-time flag into the PCR-bound TPM NVRAM, and minimize the TXT logic to just handle key selection, in preparation for GC execution. Alice will seal (in advance) key pairs for garbling Bob's inputs. Bob may then boot into TXT to receive the keys corresponding to his input. When Bob reads a key off the device (say for input bit 0), the corresponding key (for input bit 1) is erased.[7] By instantiating an OTM in this manner, we can replace interactive oblivious transfer (OT) and perform the rest of the garbled circuit execution offline, passing key output from trusted selection. By combining TXT and garbled circuits in this way, sealing complexity is now tied to Bob's inputs. We name this alternate construction *GC-based* (System 2).

[7] Unselected keys remain sealed, if never unsealed it serves as cryptographic deletion.

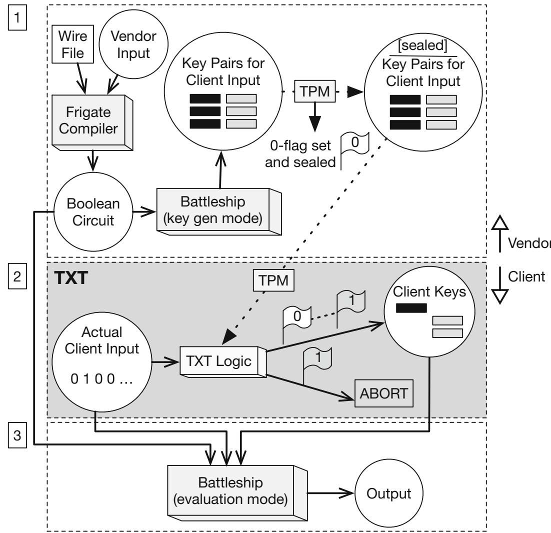

Fig. 2. In our GC-based approach to OTP, Alice generates key pairs and seals them. Bob unseals the keys that correspond to his input and locally evaluates the function.

Performance Overhead with TPM Sealing. According to our measurement, each TPM sealing/unsealing operation takes about 500 ms, and therefore 1 GB of key pairs would need about 1000 h, which is infeasible. Instead, we generate a random number as an encryption key (MK) at provisioning time and the GC key pairs are encrypted with MK. We only seal MK. This way, MK becomes per-deployment, and reprovisioning the system will not make the sealed key pairs reusable due to the change of MK (i.e., the old MK is replaced by the new key). Note that we could also apply the same approach to TXT-only (i.e., encrypting Alice's input with MK and sealing only MK), if needed by the application.

Memory Exposure. Similarly to the TXT-only OTP, our GC-based approach can also optionally adapt to address the cold-boot attack. MK becomes a single point of failure if exposed in such memory attacks, i.e., all key pairs can be decrypted and one-timeness is lost. As with TXT-only, for smaller-sized client input, we can seal the key pairs directly and only unseal into RAM in small chunks.

5.1 Implementation

We use the Boolean circuit compiler *Frigate* [32] to implement the garbled circuit components of *GC-based*. We choose the Frigate compiler for the following reasons: Frigate outperforms several other garbled-circuit compilers; it is also extensively validated and found to produce correct and functioning circuits where other compilers fail [32]. The interpreter and execution functionalities of *Frigate* are separately referred to as *Battleship*. For our purposes, we split *Battleship*

execution into two standalone phases: a key pair generation phase (`gen`) and a function evaluation phase (`evl`). Our specific modifications to *Battleship* that make split-phase execution possible are detailed in the full version of our paper.

Our GC-based approach to OTP relies on TXT for trusted key selection and leaves the computation for garbled circuits, as shown in Fig. 2. In our setting, Alice represents the vendor and Bob represents the client.

Provisioning at Alice's Site. Alice sets up the OTP box by doing the following: (1) Initialize flag to 0 and seal in the TXT program's provisioning mode. (2) Write and compile, using *Frigate*, the wire program (.wir), together with Alice's input, into the circuit.[8] (3) Load the compiled .mfrig and .ffrig files, vendor's input, and the *Battleship* executable onto the box. (4) Write the TXT program (for key selection) in the same way as in TXT-only. (5) Run *Battleship* in key-generation mode to generate the k_i^0 and k_i^1 key-pairs corresponding to each of the i bits of Bob's input. These are saved to file. (6) Seal the newly generated key pairs onto the hard-drive in provisioning mode of the TXT program. Alice is able to generate the correct number of key pairs, since garbled circuit programs take inputs of a predetermined size, meaning Alice knows the size of Bob's input. Costly sealing of all key pairs could be switched out for sealing of the master key (MK) used to encrypt the key pairs.

Evaluation at Bob's Site. Bob, upon receiving the OTP box from Alice, performs the following steps to evaluate the function on his input: (1) Place the file with Bob's input bits on the hard drive. (2) Load the TXT program in normal (non-provisioning) mode for key selection. (3) Receive selected keys corresponding to Bob's input bits; these are output to disk in plaintext. As long as it is Bob's first attempt to select keys, the TXT program will return the keys corresponding to Bob's input. Otherwise, the TXT program will abort upon loading in step (2), as shown in Fig. 2. After Bob's inputs have been successfully garbled (or converted into keys) and saved on the disk, Bob can continue with the evaluation properly. TXT is no longer required. (4) Reboot the system into the OS (e.g., Ubuntu). (5) Launch *Battleship* in circuit-evaluation mode. (6) Receive the evaluation result from *Battleship*. When *Battleship* is launched in circuit-evaluation mode, the saved keys corresponding to Bob's input are read in. *Battleship* also takes vendor input (if not compiled into the circuit) before processing the garbled circuit. The Boolean circuit is read in from the .mfrig and .ffrig files produced by *Frigate*. Evaluation is non-interactive and offline. The evaluation result is available only to Bob.

6 Case Study

We apply our proposed systems on a concrete use case based on genomic testing as a prototype. Single nucleotide polymorphism (SNP) is a common form

[8] The wire program may be written and compiled on a separate machine from that which will be shipped to Bob. If Alice chooses to use the same machine, the (no longer needed) raw wire code and *Frigate* executable should be removed from the box before provisioning continues.

of mutation in human DNA. Certain sets of SNPs determine the susceptibility of an individual to specific diseases. Analyzing an individual's set of SNPs may reveal what kind of diseases a person may have. More generally, genomic data can uniquely identify a person, as it not only gives information about a person's association with diseases, but also about the individual's relatives [35]. Indeed, advancements in genomics research have given rise to concerns about individual privacy and led to a number of related work in this space. For instance, Canim et al. [7] and Fisch et al. [10] utilize tamper-resistant hardware to analyze/store health records. Other works [2,49] investigate efficient, privacy-preserving analysis of health data.

While a number of different techniques have been proposed for privacy-preserving genomic testing, ours is the first work to address this using one-time programs grounded in secure hardware. Other than providing one-timeness, the proposed scheme also provides (i) *non-interactivity*, in which the user does not need to interact with the vendor during the protocol, and (ii) *pattern-hiding*, which ensures that the patterns used in vendor's test are kept private from the user. On the other hand, homomorphic encryption-based schemes [1] lack non-interactivity and functional encryption-based schemes [34] lack non-interactivity and pattern-hiding. We did not specifically implement these other techniques and compare our solution with them. However, from the performance results that are reported in the original papers, we can argue that our proposed scheme provides comparable (if not better) efficiency compared to these techniques.

Our aim is to prevent the adversary (the client/Bob), who uses the device for genomic testing, from learning which positions of his genome are checked and how they are checked, specifically for the genomic testing of the breast cancer (BRCA) gene. BRCA1 and BRCA2 are tumor suppressor genes. If certain mutations are observed in these genes, the person will have an increased probability of having breast and/or ovarian cancer [48]. Hence, genomic testing for BRCA1 and BRCA2 mutations is highly indicative of individuals' predisposition to develop breast and/or ovarian cancer.

We aim also to protect the privacy of the vendor (the company/Alice) that provides the genomic testing and prevent the case where the adversary extracts the test, learns how it works, and consequently, tests other people without having to purchase the test. We aim to protect both the locations that are checked on the genome and the magnitude of the risk factor corresponding to that position. Note that client's input is secure, as Bob is provided the device and he does not have to interact with Alice to perform the genomic test.

6.1 Genomic Test

In order to perform our genomic testing, we obtained the SNPs related with BRCA1[9] along with their risk factors from SNPedia [8], an open source wiki site

[9] Similarly, we can also list the SNPs for BRCA2 and determine the contribution of the observed SNPs to the total risk factor.

that provides the list of these SNPs. The SNPs that are observed on BRCA1 and their corresponding risk factors for breast cancer are omitted here for brevity.

We obtain genotype files of different people from the openSNP website [14]. The genotype files contain the extracted SNPs from a person's genome. At a high level, for each SNP of the patient that is linked to BRCA1, we add the corresponding risk factor to the overall risk.

If a BRCA1-associated SNP is observed in the patient's SNP file, we check the allele combination and add the corresponding risk factor to the total. In order to prevent a malicious client from discovering which SNPs are checked, we check every line in the patient's SNP file. If an SNP related to breast cancer is not observed at a certain position, we add zero to the risk factor rather than skipping that SNP to prevent inference of checked SNPs using side channels.

Let i denote the reference number of an SNP and s_i^j be the allele combination of SNP i for individual j. Also, S_i and C_i are two vectors keeping all observed allele combinations of SNP i and the corresponding risk factors, respectively. Then, the equation to calculate the total risk factor for individual j can be shown as $RF_j = \sum_i f(s_i^j)$ where

$$f(s_i^j) = \begin{cases} C_i(\ell) & \text{if } s_i^j = S_i(\ell) \text{ for } \ell = 0, 1, \ldots, |S_i| \\ 0 & \text{otherwise} \end{cases}$$

For instance, for the SNP with ID $i = \text{rs28897696}$, $S_i = <AA, AC>$ and $C_i = <7, 6>$. If the allele combination of SNP rs28897696 for individual j corresponds to one of the elements in S_i, we add the corresponding value from C_i to the total risk factor.

6.2 Construction for GC-Based

The garbled circuit version of the genomic test presented in Sect. 6.1 is written as wire (.wir) code accepted by the *Frigate* garbled circuit compiler. The code follows the test description in Sect. 6.1, adjusting overall risk factor upon comparing allele-pairs of matching SNPs and explicitly adding zero when needed.

We choose Bob's input from AncestryDNA files available on the openSNP website [14]. We perform preprocessing on these to obtain a compact representation of the data. Alice's input is hard-coded into the circuit at compile-time, by initializing an `unsigned int` of vendor input size and assigning each bit's value using *Frigate*'s wire operator.

Final Input Representation. Following the original design of *Battleship*, inputs are accepted as a single string of hex digits (each 4 bits). Each digit is treated separately, and input is parsed byte-by-byte (e.g., 41_{16} is represented as 10000010_2).

We use 7 hex digits (28 unsigned bits) for the SNP reference number and a single hex digit (4 unsigned bits) to represent the allele pair out of 16 possible combinations of A/T/C/G. Alice's input contains 2 more hex digits (8 signed bits) for risk factor, supporting individual risk factor values ranging from -128 to 127. We keep risk factor a signed value, since some genetic mutations lower

the risk of disease. Although we did not observe any such mutations pertaining to BRCA1, our representation gives extensibility to tests for other diseases.

Output Representation. The program outputs a signed 16-bit value, allowing us to support cumulative risk factor ranging from −32,768 to 32,767.[10]

6.3 Construction for TXT-only

In TXT-only, the genomic test logic of Sect. 6.1 is ported in pure C but largely keeps the representation used by the GC program (Sect. 6.2). Alice's input is in the form of 7 hex digits for the SNP ID, 1 hex digit for the allele pair and 2 digits for the risk factor. Bob's input is 2 digits shorter without the risk factor.

We pay special attention to minimizing exposure of Alice's input in RAM to defend against potential cold-boot attack. We achieve this by processing one record at a time performing all operations on and deleting it before moving on to the next record. We also seal each record (10 bytes) into one sealed chunk (322 bytes), which consumes more space. In each iteration, we unseal one of Alice's records and compare with all of Bob's records. For certain laptops and other computers with RAM soldered on the motherboard, this is optional.

Table 1. TXT-only results with vendor input fixed at 880 bits and varying client input size, averaged over 10 runs. Prov./Exec. refers to the provisioning mode and execution mode respectively.

Client input (bits)	Prov. (ms)	Exec. (ms)
224	5640.17	9394.58
2K	5640.17	9393.88
22K	5640.17	9388.27
224K	5640.17	9426.56
2M	5640.17	11078.19
22M	5640.17	33427.50

Table 2. TXT-only results with client input fixed at 224k bits and varying vendor input size, averaged over 10 runs. Performance of TXT-only is linear and time taken is proportional to vendor input size.

Vendor input (bits)	Prov. (ms)	Exec. (ms)
880	5640.17	9426.56
8800	53515.75	92551.43
88000	527026.89	921338.53

7 Performance Evaluation

In this section, we evaluate the two OTP systems' performance/scalability, with varying client and vendor inputs, and try to statistically verify the suitability of the two intuitive designs in different usage scenarios. We perform our evaluation

[10] This can easily be adjusted, but is accompanied by substantial changes in the resulting circuit size. For example, an 11 GB circuit that outputs 16 bits grows to 18 GB by doubling the output size to 32 bits. We conservatively choose 16 bits for demonstration purposes, but the output size may be reduced as appropriate.

on a machine with a 3.50 GHz i7-4771 CPU, Infineon TPM 1.2, 8 GB RAM, 320 GB primary hard-disk, additional 1 TB hard-disk[11] functioning as a one-time memory (dedicated to storing garbled circuit, and client and vendor input), running Ubuntu 14.04.5 LTS. In one case, we required an alternate testing environment: a server-class machine with a 40 core 2.20 GHz Intel Xeon CPU and 128 GB of RAM.[12]

We perform experiments to determine the effects of varying either client or vendor input size. Based on the case study, the vendor has 880 bits and the client has 22.4M bits of input, so we use 224 and 880 as the base numbers for our evaluation. We multiply by multiples of 10 to show the effect of order-of-magnitude changes on inputs. We start with 224 for client and 880 for vendor inputs. When varying client input, we fix vendor input at 880 bits. When varying vendor input, we fix client input at 224K bits.

7.1 Benchmarking TXT-only

Varying Client Input. Table 1 shows the timing results for TXT-only provisioning and execution with fixed vendor input and varying client input size. During provisioning, only the vendor input is sealed, so the provisioning time is constant in all cases. As client input size increases, so does execution time, but moderately. Performance is insensitive to client input size up through the 224K case. Even for the largest (22M) test case, increasing the client input size by two orders of magnitude results only in a slowdown by a factor of 3.5x.

Varying Vendor Input. Table 2 shows the timing results with fixed client input and varying vendor input size. Although we only tested against three configurations, we see an order-of-magnitude increase in vendor input size is accompanied by an order-of-magnitude increase in both provisioning and execution times.

7.2 Benchmarking GC-Based

We use the same experimental setup as used in TXT-only, but with additional time taken by the GC portion. Vendor and client each incur runtime costs from a GC (`gen`/`evl`) and a sealing-based (Prov./Sel.) phase.

Table 3. GC-based results with client input fixed at 224k bits, varying vendor input size, and encryption of keys by a sealed master key, averaged over 10 runs.

Vendor input (bits)	`gen` (ms)	Prov. (ms)	Sel. (ms)	`evl` (ms)
880	2323.7	4244.03	2508.73	31815.4
8800	3198.7	4244.03	2508.73	32200.4
88000	3286.9	4244.03	2508.73	32000.9

[11] We use a second disk to simulate what is shipped to the client (with all test data consolidated), separate from our primary disk for development.

[12] Another option would have been to upgrade the memory of the initial evaluation machine, but we chose to forgo this, as a test run on the server-class machine revealed that upwards of 60 GB would be required (not supportable by the motherboard).

Varying Vendor Input. We are interested in whether *GC-based* is less sensitive to the size of Alice's input than *TXT-only*; see Table 3. Since provisioning (Prov.) involves sealing a constant number of key pairs, and selection (Sel.) is dependent on the unsealing of these key pairs to output one key from each, there is no change. Both *Battleship* `gen` and `evl` mode timing is largely invariant, as well. Whereas System 1 performance was linearly dependent on vendor input size, we observe that *GC-based* (System 2) is indeed not sensitive to vendor input.

Table 4. GC-based results with vendor input fixed at 880 bits, varying client input size, and encryption of keys by a sealed master key, averaged over 10 runs. Provisioning- and execution-mode times were measured separately. *s indicate tests run in an alternate environment, due to insufficient memory on our primary setup.

Client input (bits)	`gen` (ms)	Prov. (ms)	Sel. (ms)	`evl` (ms)
224	1503.7	843.64	600.55	1350.8
2K	1318.9	906.70	688.62	1631.8
22K	1659.7	991.91	724.24	3643.7
224K	2323.7	4244.03	2508.73	31815.4
2M	16842.8	33934.54	19188.31	305362.8
22M	148387.9*	346606.87	283704.57	3108271*

Varying Client Input. For completeness, we also examine the effects of varying client input size on runtime; see Table 4. Prov. and Sel. stages are both slow as client input size increases, since more key pairs must be sealed/unsealed. `gen` and `evl` times are also affected by an increase in client input bits. Most notably, `evl` demonstrates a near order-of-magnitude slowdown from the 224K case to the 2M case, and the slowdown trend continues into the 22M case (despite using the better-provisioned machine to evaluate the 22M case). We indeed find that TXT-only OTP is complemented by GC-based OTP, where performance is sensitive to client input.

Table 5. Performance for TXT-only and GC-based OTP implementations of the BRCA1 genomic test, averaged over 10 runs. Vendor input is 880 bits. Client input is 22,447,296 bits. *s indicate tests run in an alternate environment, due to insufficient memory on our primary testing setup.

OTP type	Mode	Timing (ms)
TXT-only	Prov.	5640.17
	Exec.	33427.50
GC-based	`gen`	148387.9*
	Prov.	346606.87
	Sel.	283704.57
	`evl`	3108271*

7.3 Analysis

Onto our real-world genomic test (among other padded data sets for the evaluation purpose), Alice's input comprises the 22 SNPs associated with BRCA1. Each SNP entry takes up 40 bits, so Alice's input takes up 880 bits. Bob's input comprises the 701,478 SNPs drawn from his AncestryDNA file, each of which is represented with 32 bits, adding up to a total size of 22,447,296 bits. This genomic test corresponds to our earlier experiment with vendor input size of 880 bits and client input size of 22M bits.

Table 5 puts together the results for both OTP systems. Even at first glance, we see that TXT-only OTP vastly outperforms the GC-based OTP. Provisioning

is two orders of magnitude slower in GC-based OTP, and trusted selection itself is an order of magnitude slower than the entire execution mode of TXT-only OTP. `gen` and `evl` further introduce a performance hit to GC-based OTP (again, despite the fact that we evaluated this case on a better-provisioned machine). TXT-only is the superior option for our genomic application.

Choosing One OTP. We already saw in Sect. 7.1 that TXT-only OTP is less sensitive to client input, whereas we saw in Sect. 7.2 that GC-based OTP is less sensitive to vendor input. We illustrate the four cases in Table 6.

In this specific use-case of genomic testing, we are in the upper-right quadrant and thus the TXT-only OTP dominates. However, other use cases (considered in the full version of our paper) might occupy the lower-left quadrant; if so, GC-based will outperform the TXT-only OTP. What should we do if both inputs are of similar size (i.e., equally "small" or "large")? A safe bet is to stick with the TXT-only OTP. Even though GC technology continues to improve, garbled circuits will always be less efficient than running the code natively.

Table 6. Depending on the input sizes of vendor and client, one system may be preferred to the other. GC-based OTP is favorable when large vendor input is paired with small client input; TXT-only OTP otherwise.

Small vendor + Small client **TXT-only**	Small vendor + Large client **TXT-only**
Large vendor + Small client **GC-based**	Large vendor + Large client **TXT-only**

7.4 Another Use Case: Database Queries

To give an example where the vendor input can be significantly large, we may consider another potential and feasible application of our proposed OTP designs, where GC-based can outperform TXT-only. It is also in a medical setting where the protocol is between two parties, namely a company that owns a database consisting of patient data and a research center that wants to utilize patient data. The patient data held at the company contains both phenotypical and genotypical properties. The research center wants to perform a test to determine the relationship of a certain mutation (e.g., a SNP) with a given phenotype. There may be three approaches for this scenario:

1. **Private information retrieval** [9]**:** PIR allows a user to retrieve data from a database without revealing what is retrieved. Moreover, the user also does not learn about the rest of the data in the database (i.e., symmetric PIR [37]). However, it does not let the user compute over the database (such as calculating the relationship of a certain genetic variant with a phenotype among the people in the database).
2. **Database is public, query is private:** The company can keep its database public and the research center can query the database as much as it wants. However, with this approach the privacy of the database is not preserved. Moreover, there is no limit to the queries that the research center does.

As an alternative to this, database may be kept encrypted and the research center can run its queries on the encrypted database (e.g., homomorphic encryption). The result of the query would then be decrypted by the data owner at the end of the computation [23]. However, this scheme introduces high computational overhead.

3. **Database is not public, query is exposed:** In this approach, the company keeps its database secret and the research center sends the query to the company. This time the query of the research center is revealed to the company and the privacy of the research center is compromised.

In the case of GC-based, the company stores its database into the device (in the form of garbled circuit) and the research center purchases the device to run its query (in TXT) on it. This system enables both parties' privacy. The device does not leak any information about the database and also the company does not learn about the query of the research center, as the research center purchases the device and gives the query as an input to it. In order to determine the relationship of a certain mutation to a phenotype, chi-squared test can be used to determine the p-value, that helps the research center to determine whether a mutation has a significant relation to a phenotype. We leave this to future work.

8 Security Analysis

(a) Replay attacks. The adversary may try to trick the OTP into executing multiple times by replaying a previous state, even without compromising the TEE, or the one-time logic therein. The secrets (e.g., MK) only have per-deployment freshness (fixed at Alice's site). Nevertheless, in our implementation, the TPM NVRAM indices where the one-timeness flag and MK are stored are configured with PCR-bound protection, i.e., outside the correct environment, they are even inaccessible for read/write, let alone to replay.

(b) Memory side-channel attacks. Despite the hardware-aided protection from TEE, sensitive plaintext data must be exposed at certain points. For instance, MK is needed for encrypting/decrypting key pairs, and the key pairs when being selected must also be in plaintext. Software memory attacks [6,26,30] do not apply to our OTP systems, as the selected TEE (TXT) is exclusive. In our design, the code running in TEE does not even involve an OS, driver, hypervisor, or any software run-time. There are generally two categories of physical memory attacks: non-destructive ones that can be repeated (e.g., DMA attacks [38]); and the destructive (only one attempt) physical cold-boot attack [17]. All I/O access (especially DMA) is disabled for the TEE-protected regions and thus DMA attacks no longer pose a threat.

The effective cold-boot attack requires that the RAM modules are swappable and plaintext content is in RAM. For certain laptops or barebone computers [22], their RAM is soldered on the motherboard and completely unmountable (and thus immune). To ensure warm-boot attacks [47] (e.g., reading RAM content on

the same computer by rebooting it with a USB stick) are also prevented, we can set the Memory Overwrite Request (MOR) bit to signal the UEFI/BIOS to wipe RAM on the next reboot before loading any system (cf. the official TCG mitigation [44]). We do take into account the regular desktops/laptops vulnerable to the cold-boot attack: For small-sized secrets like MK, existing solutions [15,33,39,46] can be used, where CPU/GPU registers or cache memory are used to store secrets. For larger secrets, like the key pairs/vendor input, we perform block-wise processing so that at any time during the execution, only a very small fraction is exposed. Also, as cold-boot attack is destructive, the adversary will not learn enough to reveal the algorithm or reuse the key pairs. At least, the vendor can always choose computers with soldered-down RAM.

(c) Attack cost. Bob may try to infer the protected function and vendor inputs by trying different inputs in multiple instances. This attack may incur a high cost as Bob will need to order the OTP from Alice several times. This is a limitation of any offline OTP solution, which can only guarantee one query per box.

(d) Cryptographic attacks. The security of one-time programs (and garbled circuits) is proven in the original paper [12] (updated after caveat [3]), so we do not repeat the proofs here.

(e) Clonability. Silicon attacks on TPM can reveal secrets (including the Endorsement Key), but chip imaging/decapping requires high-tech equipment. Thus, cloning a TPM or extracting an original TPM's identity/data to populate a virtual TPM (vTPM) is considered unfeasible. Sealing achieves platform-state-binding without attestation, so non-genuine environments (including vTPM) will fail to unseal. We discuss TPM relay and SMM attacks in the full version of our paper. Furthermore, there has been a recent software attack [18] that resets and forges PCR values during S3 processing exploiting a TPM 2.0 flaw (SRTM) and a software bug in tboot (DRTM). They (allegedly patched) do not pose a threat to our OTP design, as neither SRTM nor any OS software (e.g., Linux) is involved, not to mention our OTP does not support/involve any power management.

9 Concluding Remarks

Until now, one-time programs have been theoretical or required highly customized/expensive hardware. We shift away from crypto-intensive approaches to the emerging but time-tested trusted computing technologies, for a practical and affordable realization of OTPs. With our proposed techniques, which we will release publicly, anyone can build a one-time program today with off-the-shelf devices that will execute quickly at a moderate cost. The cost of our proposed hardware-based solution for a single genomic test can be further diluted by extension to support multiple tests and multiple clients on a single device (which our current construction already does). The general methodology we provide can be adapted to other trusted execution environments to satisfy various application scenarios and optimize the performance/suitability for existing applications.

A Appendix

For space considerations, we also publish a full version [52] of this paper that provides additional information as follows:

- More background helpful for understanding on one-time programs, garbled circuits, and one-time memories;
- Discussion of an adaptive security attack on OTP systems;
- Detailed modifications we make to *Battleship*;
- Preprocessing steps for our case study application;
- Additional one-time program use cases;
- A list of the SNPs associated with BRCA1;
- Details of our genomic algorithm;
- Comments on porting efforts required for OTP; and
- Discussion of more attacks (e.g., SMM and TPM relay attacks).

References

1. Ayday, E., Raisaro, J.L., Laren, M., Jack, P., Fellay, J., Hubaux, J.P.: Privacy-preserving computation of disease risk by using genomic, clinical, and environmental data. In: Proceedings of USENIX Security Workshop on Health Information Technologies (HealthTech 2013). No. EPFL-CONF-187118 (2013)
2. Baldi, P., Baronio, R., De Cristofaro, E., Gasti, P., Tsudik, G.: Countering gattaca: efficient and secure testing of fully-sequenced human genomes. In: Proceedings of the 18th ACM CCS 2011, pp. 691–702 (2011)
3. Bellare, M., Hoang, V.T., Rogaway, P.: Adaptively secure garbling with applications to one-time programs and secure outsourcing. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 134–153. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34961-4_10
4. Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.R.: Software grand exposure: SGX cache attacks are practical. In: 11th USENIX Workshop on Offensive Technologies (WOOT 2017), Vancouver, BC (2017)
5. Broadbent, A., Gutoski, G., Stebila, D.: Quantum one-time programs. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 344–360. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40084-1_20
6. Bulck, J.V., et al.: Foreshadow: extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In: USENIX Security Symposium, Baltimore, MD, USA, pp. 991–1008 (2018)
7. Canim, M., Kantarcioglu, M., Malin, B.: Secure management of biomedical data with cryptographic hardware. IEEE Trans. Inf Technol. Biomed. **16**(1), 166–175 (2012)
8. Cariaso, M., Lennon, G.: SNPedia: a wiki supporting personal genome annotation, interpretation and analysis (2010). http://www.SNPedia.com
9. Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: Proceedings of the 36th Annual Symposium on Foundations of Computer Science, pp. 41–50. IEEE (1995)
10. Fisch, B.A., Vinayagamurthy, D., Boneh, D., Gorbunov, S.: Iron: functional encryption using Intel SGX. Technical report, IACR eprint (2016)
11. Gnu.org: The multiboot specification (2009). http://www.gnu.org/software/grub/manual/multiboot/multiboot.html

12. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: One-time programs. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 39–56. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85174-5_3
13. Greene, J.: Intel® trusted execution technology. Technical report (2012)
14. Greshake, B., Bayer, P.E., Rausch, H., Reda, J.: Opensnp-a crowdsourced web resource for personal genomics. PLoS ONE **9**(3), 1–9 (2014)
15. Guan, L., Lin, J., Luo, B., Jing, J.: Copker: computing with private keys without RAM. In: NDSS, San Diego, CA, USA, February 2014
16. Gunupudi, V., Tate, S.R.: Generalized non-interactive oblivious transfer using count-limited objects with applications to secure mobile agents. In: Tsudik, G. (ed.) FC 2008. LNCS, vol. 5143, pp. 98–112. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85230-8_8
17. Halderman, J.A., et al.: Lest we remember: cold boot attacks on encryption keys. In: USENIX Sec 2008, San Jose, CA, USA (2008)
18. Han, S., Shin, W., Park, J.H., Kim, H.: A bad dream: subverting trusted platform module while you are sleeping. In: 27th USENIX Security Symposium (USENIX Security 2018), Baltimore, MD, USA, pp. 1229–1246 (2018)
19. Hazay, C., Lindell, Y.: Efficient Secure Two-Party Protocols. ISC. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14303-8
20. Intel Corporation: Trusted boot (tboot), version: 1.8.0 (2017). http://tboot.sourceforge.net/
21. Järvinen, K., Kolesnikov, V., Sadeghi, A.-R., Schneider, T.: Garbled circuits for leakage-resilience: hardware implementation and evaluation of one-time programs. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 383–397. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15031-9_26
22. Jefferies, C.P.: How to identify user-upgradeable notebooks, June 2017. http://www.notebookreview.com/feature/identify-user-upgradeable-notebooks/
23. Kantarcioglu, M., Jiang, W., Liu, Y., Malin, B.: A cryptographic approach to securely share and query genomic sequences. IEEE Trans. Inf Technol. Biomed. **12**(5), 606–617 (2008)
24. Kirkpatrick, M.S., Kerr, S., Bertino, E.: PUF ROKs: a hardware approach to read-once keys. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, AsiaCCS 2011, Hong Kong, China, pp. 155–164 (2011)
25. Kitamura, T., Shinagawa, K., Nishide, T., Okamoto, E.: One-time programs with cloud storage and its application to electronic money. In: APKC (2017)
26. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. CoRR (2018)
27. Kollenda, B., Koppe, P., Fyrbiak, M., Kison, C., Paar, C., Holz, T.: An exploratory analysis of microcode as a building block for system defenses. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, 15–19 October 2018, pp. 1649–1666 (2018)
28. Koppe, P., et al.: Reverse engineering x86 processor microcode. In: 26th USENIX Security Symposium (USENIX Security 2017), Vancouver, BC, pp. 1163–1180 (2017)
29. Lee, S., Shih, M.W., Gera, P., Kim, T., Kim, H., Peinado, M.: Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In: 26th USENIX Security Symposium (USENIX Security 2017), Vancouver, BC, pp. 557–574 (2017)
30. Lipp, M., et al.: Meltdown. CoRR (2018)
31. McCune, J.M.: Reducing the trusted computing base for applications on commodity systems. Ph.D. thesis, Carnegie Mellon University (2009)

32. Mood, B., Gupta, D., Carter, H., Butler, K., Traynor, P.: Frigate: a validated, extensible, and efficient compiler and interpreter for secure computation. In: Euro-SP (2016)
33. Müller, T., Freiling, F.C., Dewald, A.: TRESOR runs encryption securely outside RAM. In: USENIX Security Symposium, San Francisco, CA, USA, August 2011
34. Naveed, M., et al.: Controlled functional encryption. In: CCS 2014, pp. 1280–1291. ACM (2014)
35. Naveed, M., et al.: Privacy and security in the genomic era. In: CCS 2014 (2014)
36. nccgroup: Cachegrab, December 2017. https://github.com/nccgroup/cachegrab
37. Saint-Jean, F.: Java implementation of a single-database computationally symmetric private information retrieval (cSPIR) protocol. Technical report, Yale University Department of Computer Science (2005)
38. Sevinsky, R.: Funderbolt: Adventures in Thunderbolt DMA Attacks. Black Hat USA (2013)
39. Simmons, P.: Security through Amnesia: a software-based solution to the cold boot attack on disk encryption. In: ACSAC (2011)
40. Sottek, T.: NSA reportedly intercepting laptops purchased online to install spy malware, December 2013. https://www.theverge.com/2013/12/29/5253226/nsa-cia-fbi-laptop-usb-plant-spy
41. Strackx, R., Jacobs, B., Piessens, F.: ICE: a passive, high-speed, state-continuity scheme. In: Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, Louisiana, USA, pp. 106–115 (2014)
42. Strackx, R., Piessens, F.: Ariadne: a minimal approach to state continuity. In: 25th USENIX Security Symposium (USENIX Sec 2016), Austin, TX, pp. 875–892 (2016)
43. Tarnovsky, C.: Attacking TPM part 2: a look at the ST19WP18 TPM device, July 2012. dEFCON presentation. https://www.defcon.org/html/links/dc-archives/dc-20-archive.html
44. Trusted Computing Group: TCG Platform Reset Attack Mitigation Specification, May 2008
45. Trusted Computing Group: Trusted Platform Module Main Specification, version 1.2, revision 116 (2011). https://trustedcomputinggroup.org/tpm-main-specification/
46. Vasiliadis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.: PixelVault: using GPUs for securing cryptographic operations. In: CCS 2014, Scottsdale, AZ, USA, November 2014
47. Vidas, T.: Volatile memory acquisition via warm boot memory survivability. In: 43rd Hawaii International Conference on System Sciences, pp. 1–6, January 2010
48. Walsh, T., et al.: Detection of inherited mutations for breast and ovarian cancer using genomic capture and massively parallel sequencing. Natl Acad. Sci. **107**(28), 12629–12633 (2010)
49. Wang, X.S., Huang, Y., Zhao, Y., Tang, H., Wang, X., Bu, D.: Efficient genome-wide, privacy-preserving similar patient query based on private edit distance. In: CCS, pp. 492–503. ACM (2015)
50. Yao, A.C.: Protocols for secure computations. In: FOCS (1982)
51. Zhang, N., Sun, K., Shands, D., Lou, W., Hou, Y.T.: Truspy: cache side-channel information leakage from the secure world on ARM devices. IACR Cryptology ePrint Archive 2016, 980 (2016)
52. Zhao, L., et al.: One-time programs made practical (2019). http://arxiv.org/abs/1907.00935