ICE: A Passive, High-Speed, State-Continuity Scheme (Extended Version)

Raoul Strackx
iMinds-Distrinet, KU Leuven
Celestijnenlaan 200A
3001 Heverlee

Bart Jacobs
iMinds-Distrinet, KU Leuven
Celestijnenlaan 200A
3001 Heverlee
first.last@cs.kuleuven.be

Frank Piessens
iMinds-Distrinet, KU Leuven
Celestijnenlaan 200A
3001 Heverlee

ABSTRACT

The amount of trust that can be placed in commodity computing platforms is limited by the likelihood of vulnerabilities in their huge software stacks. Protected-module architectures, such as Intel SGX, provide an interesting alternative by isolating the execution of software modules. To minimize the amount of code that provides support for the protected-module architecture, persistent storage of (confidentiality and integrity protected) states of modules can be delegated to the untrusted operating system. But precautions should be taken to ensure *state continuity*: an attacker should not be able to cause a module to use stale states (a so-called *rollback attack*), and while the system is not under attack, a module should always be able to make progress, even when the system could crash or lose power at unexpected, random points in time (i.e., the system should be *crash resilient*).

Providing state-continuity support is non-trivial as many algorithms are vulnerable to attack, require on-chip non-volatile memory, wear-out existing off-chip secure non-volatile memory and/or are too slow for many applications.

We introduce ICE, a system and algorithm providing state-continuity guarantees to protected modules. ICE's novelty lies in the facts that (1) it does not rely on secure non-volatile storage for every state update (e.g., the slow TPM chip). (2) ICE is a passive security measure. An attacker interrupting the main power supply or any other source of power, cannot break state-continuity. (3) Benchmarks show that ICE already enables state-continuous updates almost 5x faster than writing to TPM NVRAM. With dedicated hardware, performance can be increased 2 orders of magnitude.

We present a machine-checked proof of ICE's security guarantees and evaluate a prototype implementation on commodity hardware.

1. INTRODUCTION

Protection of sensitive data in commodity computing platforms is extremely challenging. Modern operating systems provide process isolation primitives, but the kernel itself is too large to be implemented free from vulnerabilities. Moreover the operating system's functionality is extended continuously to support new file systems, process scheduling algorithms, peripherals, etc. Commodity systems are also prone to physical attacks, even by ill-equipped and resource-constrained home users [9, 14]. These vulnerabilities limit the amount of trust that can be placed in commodity systems. In servers these limitations are remedied by programmable hardware security modules (HSMs). On client devices, highly-sensitive applications such as online banking or e-government often resort to smart cards. Unfortunately, these solutions are expensive, cumbersome and the security guarantees that they can provide to the overall applications are limited.

Two recent advances in computer security indicate that this situation may change in the near future. First, protected-module architectures (PMAs) have been developed that provide strong isolation directly to modules running at application level [4, 8, 24, 25, 27, 33, 37, 38, 45]. The OS is still relied upon to provide services such as disk and network access, but they are *not* trusted. Protected modules' memory regions cannot be accessed from unprotected memory; modules are in complete control over their own content and can only be accessed through the interface they expose. Last year Intel disclosed their work on Software Guard eXtension (SGX) [3,15,26], their own hardware-implemented protected module architecture for commodity processors. SGX goes even further than other state-of-the-art protected-module architectures and also provides protection against hardware attacks; modules (called enclaves in SGX[1]) are only stored in plaintext within the CPU package. When they are evicted to main memory they are confidentiality, integrity and version protected.

Second, Agten et al. [2] and Patrignani et al. [31] proposed fully-abstract compilation techniques to such protected module architectures. While the strong isolation guarantees offered by these architectures is vital, they are difficult to implement without compiler support. Care must be taken not to introduce software vulnerabilities during compilation. Fully-abstract compilation ensures just this; machine-code-level attacks exists *iff* also a corresponding attack at source-code level exists. This enables easy reasoning and verification of the security guarantees these modules provide.

Unfortunately an important attack vector has been largely overlooked. Protected-module architectures, including SGX, only provide strong isolation guarantees *while the system executes continuously*. Without support for state continuity, protected modules need to remain stateless, significantly hampering their applicability. Consider as a running example a password-checking module. To defend against dictionary attacks, the user will be locked out indefinitely after three failed attempts. The module confidentiality and integrity protects its state before handing it to the untrusted operating system for storage. But when the module needs

[1]We will use the term "protected module" when referring to isolated memory areas in *any* protected-module architecture and use "enclave" when referring to SGX specifically.

to recover its state after a reboot, it cannot distinguish between a fresh and a stale state and the guess-limited security measure cannot be guaranteed.

While having similarities with replay attacks at first glance, the state itself is replayed in a rollback attack. Providing support for state continuity is therefore much harder, especially when practical limitations are considered. Parno et al. [30] show that many seemingly obvious algorithms are flawed. Others are prone to simple hardware attacks. Attaching an uninterruptible power source (UPS), for example, may simply be disconnected. Or an in-kernel attacker may prevent the execution of the interrupt handlers it relies upon. Adding non-volatile memory on-chip could simplify a solution, but requires modification of manufacturing processes leading to increased manufacturing costs. Alternatively, using non-volatile memory off-chip (e.g., isolating disk space) may be susceptible to a clone attack where a hardware-level attacker may easily overwrite the state with a previously recorded stale state. Using TPM NVRAM or TPM monotonic counters instead, would foil such attacks, but would significantly impact performance and usability. Most implementations only provide 1,280 bytes of NVRAM that supports only 100,000 write cycles over the chip's lifetime [30]. Accessing NVRAM every second, would wear it out in less then 28 hours. Monotonic counters, on the other hand, only need to be incrementable every 5 seconds [44].

Hardware upgrades to the TPM chip could reduce some of these architectural constraints, at an economic cost. However, any solution placing the TPM on the performance-critical path, would require additional upgrades over time to bridge the ever growing TPM/CPU performance gap. We present ICE, an alternative solution that only requires TPM accesses at boot time and is thus *not* affected by TPM speed.

ICE avoids architectural challenges (1) by proposing a simple implementation technique where on-chip dedicated registers are backed off-chip by a capacitor and persistent memory. Upon a sudden loss of power, the contents of the dedicated registers is written to persistent memory. (2) ICE is a *passive* protection scheme; in the event of a crash or power loss, security is guaranteed instantly. A hardware attacker may disconnect the capacitor, but state continuity remains guaranteed. (3) At the moment freshness information is backed to persistent storage, it is considered public data. Overwriting it with stale freshness information will be detected upon recovery.

In summary, we make the following contributions:

- We present ICE, the first algorithm providing state-continuity guarantees with a minimal TCB that does *not* rely on the speed of secure, non-volatile memory (e.g., the (slow) TPM chip) nor does it rely on an uninterruptible power source.
- We formally verify and machine check the security properties of ICE using the Coq proof assistant.
- Because SGX-enabled machines or emulators are not yet available, we validate our claims based on a prototype implementation on top of Fides [37], an existing hypervisor-based protected module architecture similar to SGX. Benchmarks show that states can already be stored almost 5x faster on commodity hardware than writing to TPM NVRAM. Dedicated hardware support would increase performance substantially.
- We provide new insights that can steer the future design of hardware security modules e.g., the TPM.

The remainder of this paper is structured as follows. First we detail our attack model and the security properties that we need to guarantee. Next in Sections 3 and 4, we present our algorithm and discuss three possible implementations. Finally, we evaluate the security and performance of ICE and discuss how it can affect future directions of hardware security modules.

2. PROBLEM DEFINITION

2.1 Attacker Model

ICE can defend against an attacker with three powerful capabilities. First, we assume that an attacker is able to compromise the entire software stack, with the exception of ICE-implementing modules. This enables versatile attacks ranging from modifying the contents of the hard drive to preventing enclaves from ever resuming execution.

Second, we assume that an attacker has control over the system's power supply or is able to launch attacks leading to a similar result. Power-interruption attacks differ from kernel-level crashes as they also affect software modules executing in complete isolation from the rest of the system: modules may stop executing before they can commit their new state. SGX enclaves are especially vulnerable to such attacks. In order to prevent denial-of-service attacks by malicious enclaves that never return control to the kernel, SGX supports interruption of enclaves [17]. When the interrupt is handled in the untrusted kernel, an in-kernel attacker can easily prevent the enclave from ever resuming execution.

Third, we consider hardware attacks. We implement ICE as a library that modules can be statically linked with and take advantage of the security guarantees provided by the protected-module architecture. In case of SGX this implies that an attacker may place probes on memory buses or perform cold boot attacks [14]. Defending against physical attacks against the CPU package itself or the TPM chip [36,41,47] are orthogonal problems and not considered.

With respect to cryptographic capabilities of the attacker, we assume the standard Dolev-Yao model [11]: cryptographic messages can be manipulated, for instance by duplicating, re-ordering or replaying them, but the underlying cryptographic primitives cannot be broken.

We do not consider side-channel attacks in general (e.g., attacks based on cache behavior [28]) but similar to Parno et al. [30] make one exception: we do consider attacks where an attacker prevents the module from recording its new state when she is able to infer (e.g., based on timing differences) that this state would be unpreferable. When input is given to a module, it should either complete the computation or no valuable information should be deducible from it.

2.2 Security Properties

State continuity can be factored into two properties: safety and liveness. To ensure safety, ICE must be resilient against a *rollback attack* where an attacker provides the module with a valid, but stale state. A rollback attack is related to a replay attack but it is much harder to defend against. Where in a replay attack identical input is provided, the state of the module itself is replayed in a rollback attack.

The second property, liveness, states that benign events

should never force the system into a state from which it cannot progress. In practice this means that the system should be allowed to crash at any time during the operation of the algorithm, including when it is recovering from a previous crash. Note that this is not the same as protection against denial-of-service. Protection against denial-of-service is not in scope; in-kernel attackers can easily prevent the system from progressing (e.g., by removing the fresh state from disk, or by breaking the kernel altogether). Liveness only ensures progress is not hampered by *random* crashes. This is important, since random crashes (or power loss) may occur even when a system is not under attack.

2.3 Applicability

ICE's high-speed state-continuity guarantees enable a large range of applications:

Fortified Applications.

Almost all non-trivial applications need to keep some kind of state: login credentials must not be rolled back to a stale state, firewall settings must not be revertible and systems must be able to prove that stored log files are fresh and have never been tampered with. State-continuity guarantees can also enable more privacy friendly applications. Many use cases (e.g., road pricing or smart electricity meters) require sensitive data to be collected and send to a remote server. With strong hardware isolation, attestation and state-continuity guarantees, a (possibly malicious) user can download a software module that collects sensitive data and only sends an aggregated value to a remote party. Privacy sensitive data does never have to leave the user's system. We will discuss in Section 7 how low-end devices can also benefit from ICE.

A Building Block for Protocols.

High-performance, state-continuous storage enable protocols to provide stronger security guarantees. Consider distributed algorithms as an example. Fault-tolerant algorithms have been proposed to reduce the impact of failing network participants (e.g., they may crash, process inputs incorrectly or their local state may get corrupted), but there is a theoretical upper bound that at most one third of the participants may be faulty. Chun et al. proposed append-only memory (A2M) [10] to harden existing distributed algorithms and applications such as NFS. Acting as a trusted log, this memory protects against equivocation; the ability of a network node to make contradicting statements to different entities. The authors propose hardened versions of PBFT [7], SUNDR [22] and Q/U [1], but leave implementation with a small TCB as future work. ICE is able to implement fast, append-only memory almost trivially.

Avoiding the TPM Chip as a Bottleneck.

Many applications and protocols could also be implemented based on guarantees provided by the TPM chip [30]. Unfortunately the TPM was never designed with performance as a main requirement and a wide application of this approach would result in a severe bottleneck, especially in a server setting where each client connection requires TPM access. ICE avoids this bottleneck and other TPM constraints; a virtually unlimited number of monotonic counters of variable length can be provided, almost unlimited, never wearing-out NVRAM can be offered, etc.

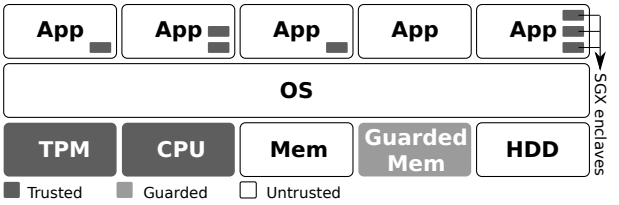

Figure 1: ICE provides state-continuity guarantees to isolated modules while trusting only a few key components.

3. STATE-CONTINUITY AS A LIBRARY

Before introducing a running example and describing ICE in full detail, we first introduce the system hardware we rely on and discuss how freshness information is recorded.

3.1 Architecture

Assuming ICE is implemented on top of Intel SGX, we only need to place trust in the CPU package and TPM chip (see Figure 1). Attacks against any other component cannot compromise security.

Enclaves.

Intel SGX, as any other protected-module architecture [4, 8, 24, 25, 27, 33, 37, 38, 45], provides enclaves with total control over their own code and data by enforcing a specific access control mechanism; *only* when executing within the boundaries of an enclave can its content be accessed. Access attempts from code running at *any* privilege level outside the enclave (including from other enclaves), will be blocked. Enclaves can only be accessed through an interface they expose explicitly.

SGX can also make hardware attacks against enclaves significantly more challenging by ensuring that their content is only stored in plaintext inside the CPU package. The operating system may choose to write pages of enclaves to RAM memory or swap disk, but only after they are confidentiality and integrity protected. Freshness data is included as well to ensure that no stale pages can be swapped back in. However, when the system shuts down or goes into hibernation or sleep mode, enclaves are destroyed and this freshness information is lost [17]. Since enclaves live in the same address space and maintain their state between invocations, they can be seamlessly integrated in applications.

TPM.

We store long term secrets and freshness information in TPM NVRAM. These secrets should only be accessible from the SGX enclave that provided them.

Guarded Memory.

To enable fast state updates, we propose the addition of a small amount of *guarded memory*; dedicated registers on-chip that are backed off-chip by shadow, non-volatile memory (NVRAM) and a capacitor (see Figure 2). When a controller detects that the main power supply is disconnected from the CPU package, it copies the registers' content to non-volatile memory. When power is re-applied, the registers are restored. Note that *only* on-chip components need to be trusted. Attacks against shadow memory, main power supply or the capacitor cannot break state continuity.

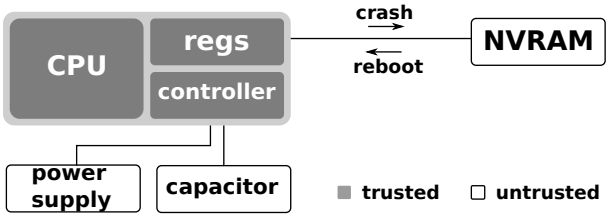

Figure 2: Architecture of guarded memory. When power suddenly fails on-chip dedicated registers are backed up to off-chip, shadow memory (NVRAM). *Only* on-chip components need to be trusted. Hardware attacks against NVRAM, main power supply or the capacitor cannot break state continuity.

The controller must also guarantee that guarded memory can be used to store sensitive data in a way that is inaccessible to an attacker. This is achieved by implementing an exclusive access mechanism. At boot time guarded memory is publicly accessible. The first enclave that requests exclusive access will receive it until the next reboot. From then on only that enclave can access guarded memory. Access requests from other enclaves or unprotected memory will be blocked. When power goes down, exclusive access is lost and data stored in guarded memory must be considered as being public.

Persistent Storage.
ICE uses operating system services to access persistent storage. These services are not trusted: an attacker may copy, replace and destroy files. To differentiate between the actual state of a module and states stored on disk, we call the latter *(ICE) cubes* whenever ambiguity might arise.

3.2 Guards: Storing Freshness Info

Just as message authentication codes (MACs) can be used to guarantee message integrity, we will use *guards* to prove that a cube is fresh. Guards are 2-tuples:

$$\text{guard}_i(n) = (\underbrace{\text{Hash}^i(n)}_{\text{guard value}}, \underbrace{i}_{\text{guard index}})$$

where the first element, the *guard value* represents the hash value after hashing the base value i times, the *guard index*.

A guard is incremented by hashing the guard value and incrementing the index:

$$\text{guard}_i(n) = (\text{Hash}^i(n), i)$$
$$\text{guard}_{i+1}(n) = (\text{Hash}^{i+1}(n), i+1)$$

Based on the construction of guards, they possess two important properties: (1) two guards can be compared based on the guard index:

$$(n,i) \leq (m,j) \Leftrightarrow \begin{cases} n = m & \text{if } i = j \\ (\text{Hash}(n), i+1) \leq (m,j) & \text{if } i < j \end{cases}$$

and, (2) an attacker is unable to calculate any preceding guard as this would imply inverting the hash function.

3.3 ChkPassword: A Running Toy Example

Guaranteeing state-continuity is non-trivial and can only be accomplished by a module provider taking the required safety precautions. We only provide a library offering state-continuous storage. To demonstrate the subtle vulnerabilities that need to be resolved, consider as a running toy example `ChkPassword`, a password-checking module displayed in listing 1. It exposes an interface of two functions: `set_passwd` that modifies the user's password and `check_passwd`[2] that handles login attempts. To prevent dictionary attacks, `ChkPassword` will lock out a user indefinitely after 3 incorrect attempts. We assume that when the module is created, the `INIT` function is called before any service call is handled. When `ChkPassword` executes on the platform for the first time, a default password is selected (line 7), otherwise its previous state is restored (line 10).

```
1  static int attempts_left;
   static char *password;
3
   void INIT( void ) {
5    State *state;
     if (retrieve( &state ) == UNINITIALIZED){
7        password = "default";
         attempts_left = 3;
9    } else
         restore_and_restart( state );
11 }
13 int ENTRY_POINT check_passwd(char *guess) {
     State *state = new State();
15
     //store (input, state) tuple
17   collect_state( state );
     collect_input( state, "CHECK_PASSWD" );
19   collect_input( state, guess );
     store( state );
21
     //check passwd
23   if ( attempts_left > 0 &&
          strcmp( password, guess ) == 0 ) {
25     attempts_left = 3;
       return OK;
27   } else {
       attempts_left = max(attempts_left -1,0);
29     return INCORRECT;
     }
31 }
33 int ENTRY_POINT set_passwd(char *oldpwd,
       char *newpwd) { ... }
```

Listing 1: ChkPassword: A running example

To ensure state continuity, `ChkPassword` needs to fulfill three requirements. First, it must protect against subtle timing attacks. When an attacker is able to infer that the provided password is incorrect based on timing differences between a correct and incorrect password[3], she may be able to crash the system before the login attempt could be recorded. Ensuring that each execution path takes exactly the same amount of CPU cycles is hard. Similar to Parno

[2]Calling `ChkPassword` from unprotected memory would enable an attacker to intercept the provided password before it reaches the module. Users of `ChkPassword` should establish a secure channel from another module before exchanging sensitive data [3, 37]. Such considerations are out of scope.

[3]A similar attack exists when a (unique) callback to unprotected memory is made before an undesirable state is stored.

et al. [30], we take a much simpler approach and store the state with the newly provided input *before* it is used in any computation. Hence, `ChkPassword` stores its current state (the number of attempts left and the correct password) together with the provided guess (line 17-20) before checking the provided password. An unexpected crash while the password is being verified (i.e., after line 20), will then result in the current state being restored and execution is restarted; another attempt is made to check the *same* provided password. We assume `restore_and_restart` restores the current state and restarts execution of the last called entry point (line 10). Alternatively, if the system crashed before the input could be recorded and thus was never used in any meaningful computation (i.e., before line 20), the guess can simply be discarded.

Second, in order to guarantee that re-execution of the same input on the same state always leads to an identical result, modules must be deterministic. This implies that modules must consider all sources of non-determinism (e.g., the result of a random number generator) as input and thus store such data before using it in any computation.

Third, an attacker must not be able to infer any value from the size of the stored states on disk; modules must ensure that all cubes are equal in size.

3.4 ICE Libraries

We will provide state-continuous storage in two steps. In Section 3.4.1 we introduce `libice0`, a library providing support at the cost of scarce platform resources for every instance. Then in Section 3.4.2, we present `libicen` that alleviates resource pressure by storing freshness information in a single, state-continuous module `ice0`. As all `libicen` library instances connect to the same, unique `ice0` instance, a virtually unlimited number of modules is supported.

Both `libice0` and `libicen` provide the same interface: `store(State *)` and `retrieve(State **)`. To avoid repeated TPM or `ice0` accesses, `libice0` and `libicen` keep a cached copy. In order to distinguish between these copies and explicitly state where they are stored, we will reference them similarly to fields of a struct. For example, the encryption and MAC keys stored in the TPM chip will be referenced as `tpm.keys`. The variables used by the ICE algorithm are referenced as `ice.keys` and so on. Besides storing keys and the guard we also keep track of the state of the algorithm using a `mode` variable. Stored inside the TPM chip (`tpm.mode`), this variable indicates whether ICE was once initiated correctly. In `libice0` (`ice.mode`) this variable is used to indicate whether ICE was initiated or recovered since reboot. We assume that when a module is resurrected after a crash, `ice.mode` is initialized with value `Clear`. As a shorthand, we also assume that setting this variable takes exclusive access of guarded memory. Listing 2 uses these variables to differentiate between an initial state being stored and a state being updated. Similarly, `tpm.mode` is used to determine whether a state was ever stored.

3.4.1 `libice0`: State-Cont. Storage for One Module

In order to provide state continuity, we must guarantee that an attacker is not able to fabricate recorded states (called cubes) and that no stale cubes can be provided as being fresh. The former is trivially guaranteed by including a message authentication code in each cube (see Fig. 3). Guaranteeing freshness is more challenging, but as modules

```
1  void store( State *state ) {
     switch (ice.mode) {
3      case Clear:
         return _init_state( state );
5      case Activated:
         return _update_state( state );
7  } }

9  int retrieve( State **state ) {
     switch (tpm.mode) {
11     case Clear:
         return UNINITIALIZED;
13     case Activated:
         *state = _recovery_step();
15       return RECOVERED;
   } }
```

Listing 2: `libice0` relies on `tpm.mode` and `ice.mode` to distinguish between storing an initial state, updating a stored state and recovery



Figure 3: Stored states are confidentiality and integrity protected. Freshness is based on the enclosed guard.

maintain their state between invocations, we only need to consider power off and reboot events. Let's call events during such power cycles an *execution stream*. An execution stream starts by either storing an initial state of a module or when the state of a module is recovered after a crash. It ends when the system crashes or when it is shut down properly.

To keep track of the fresh cube, we will generate a (base) guard when the execution stream starts and store it securely in TPM NVRAM. For every state the module requests storage of in the current execution stream, we will increment the guard and include it in the generated cube. Using guarded memory we will ensure that *only* the guard included in the last (and thus fresh) cube is leaked at the moment the system crashes. As no preceding guards were leaked (and cannot be calculated), it serves as a pointer to the fresh cube. Upon recovery, knowledge of the guard[4] that is stored in the provided cube, proves that the cube is fresh.

Creation of an Initial State.

When storage of the initial state of the module is requested, a new base guard and keys are generated (see listing 3). Next, a new cube is constructed and written to disk. Exclusive access of guarded memory is taken by setting the `ice.mode` variable to `Activated` and the fresh guard is written to guarded memory. In case exclusive access cannot be assigned (i.e., another module already received it), the module simply stops its execution. For clarity, such error handling is not displayed. Finally the keys and guard are stored in the TPM's NVRAM and `tpm.mode` is set to `Activated`, committing the start of a new execution stream.

[4]We must also check that this guard was created during the last execution stream as a matching guard/cube is found at the end of every execution stream.

```
void _init_state( State *state ) {
  ice.guard = gen_guard();
  ice.keys = gen_keys();
  hdd.write( new Cube( ice.guard, ice.keys,
       state ));
  ice.mode = Activated;
  gmem.guard = ice.guard;
  tpm.guard = ice.guard;
  tpm.keys = ice.keys;
  tpm.mode = Activated;
}
```

Listing 3: `libice0`: Storing the initial state

```
void _update_state( State *state ) {
  ice.guard = ++ice.guard;
  hdd.write( new Cube( ice.guard, ice.keys,
       state ));
  gmem.guard = ice.guard;
}
```

Listing 4: `libice0`: Updating a state

Updating a State.

When storage of a new input-state pair is requested in the same execution stream, the previously used guard and keys are still stored in `libice0`'s memory and no TPM accesses are required. To safely store the input-state pair, a new cube is created with the subsequent guard and stored on disk (see listing 4). Finally the step is committed by writing the fresh guard to guarded memory.

Recovering from a Crash.

Recovering from a crash is more challenging and is achieved in two steps (see listing 5, error handling is omitted for clarity). First, the last stored cube is read from disk. By verifying three properties its freshness is ensured:

- *Validity*: Cubes must not have been forged by an attacker. This is ensured by the MAC stored in each cube and the accompanying key stored securely in the TPM chip (line 17).
- *Correct execution stream*: The cube received from the untrusted OS must have been created during the last execution stream. At the start of each execution stream a new base guard is generated and stored in TPM NVRAM. All guards used during this execution stream are successors of this base guard. Hence, the cube was created during the last execution stream iff (line 18):

$$\text{tpm.guard} \leq \text{cube.guard}$$

- *Public guard*: `libice0` ensures that guarded memory always contains the same guard as the last (fresh) cube stored on disk[5], and that no preceding guards leak or can be calculated. Hence, if the guard stored in guarded memory matches the guard included in the cube at hand and the two previous properties hold as well, it is guaranteed that the cube is fresh (line 19).

[5]There is one exception as writing cubes to disk and updating guarded memory cannot be executed atomically. This exception is resolved later in this section.

```
State *_recovery_step() {
  Cube cube = hdd.read();
  if ( is_fresh( &cube ) ) {
    State *state = extract( cube, tpm.keys
        );
    ice.guard = gen_guard();
    ice.keys = tpm.keys;
    hdd.write( new Cube( ice.guard, ice.
        keys, state ) );
    ice.mode = Activated;
    gmem.guard = ice.guard;
    tpm.guard = ice.guard;
    return state;
  }
  else abort();
}

bool is_fresh( Cube *cube ) {
  return ( check_mac( cube, tpm.keys ) &&
    tpm.guard ≤ cube->guard &&
    gmem.guard.value == cube->guard.value);
}

bool operator≤( Guard g1, Guard g2 ) {
  while ( g1.index < g2.index ) {
    g1.value = Hash( g1.value );
    ++g1.index;
  }
  return g1.value == g2.value;
}
```

Listing 5: `libice0`: Recovering from a crash.

In the second step the fresh state is re-stored as part of a new execution stream: `libice0`'s variables are restored from TPM NVRAM, a new base guard is generated, the fresh state packaged in a new cube and the base guard is written to guarded and TPM NVRAM memory. To ensure that after an unexpected crash during the execution of this step, recovery can be restarted, `libice0` must (1) backup the previous fresh guard before overwriting it in guarded memory. As this value is public, any persistent storage can be used (for clarity not displayed in listing 5). (2) The new base guard is written to TPM NVRAM as the last step.

Let's reconsider `ChkPassword` and discuss how crashes are resolved. Depending on the timing of a crash, we can differentiate between three main situations. Fig. 4 displays them graphically. One, `ChkPassword` was just created and the user called `set_passwd` to change the default password. This led to the execution of `_init_state` but the system crashes before `tpm.mode` could be set (see listing 3 line 9, t_0 in Fig. 4). When `ChkPassword` is re-created, it requests its previous state (listing 1, line 6). As `tpm.mode` still read `Clear` (listing 2 line 3), the module will restart from its default settings. As no input was ever used, state-continuity is guaranteed trivially.

Two, the system didn't crash when the user modified the module's default password and now calls `check_passwd` providing `"attempt1"` as password. After `libice0` stores a new cube $C_{\texttt{attempt1}}$ on disk and updates guarded memory, the system crashes while the password is being verified (listing 1 line 23, t_1 in Fig. 4). The module is re-created and execution flow eventually executes `_recovery_step` (listing 5) As only a single cube is available containing the leaked guard from guarded memory (or a successor thereof), only cube $C_{\texttt{attempt1}}$

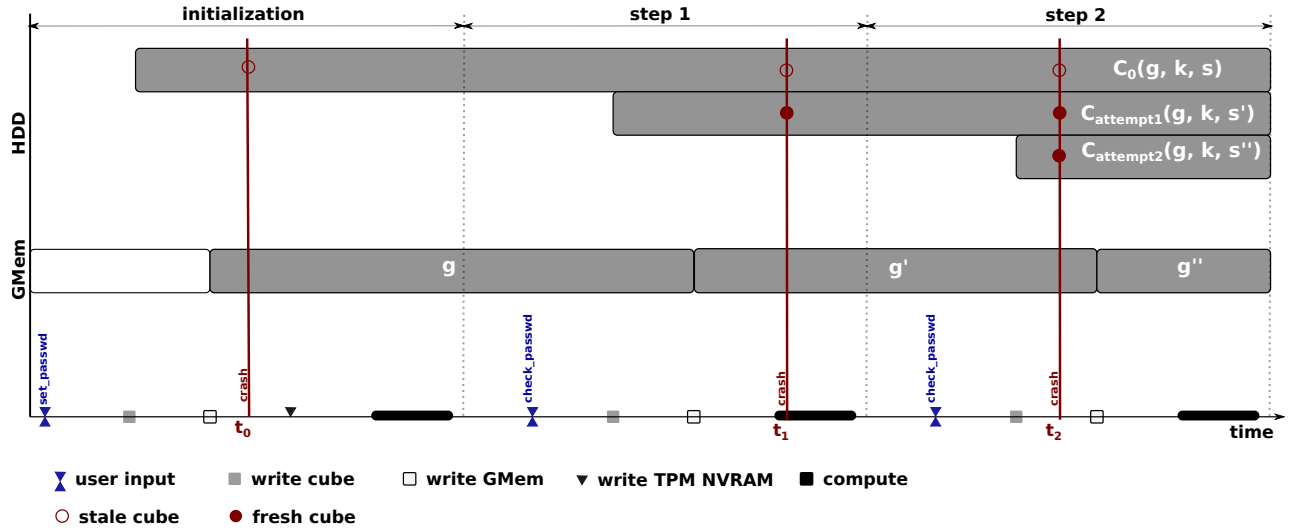

Figure 4: Graphical overview of the steps taken by a module storing its initial state and taking two steps in the same execution stream. Depending on the timing of a crash, three distinct situations can occur.

is considered fresh. After returning the stored input-state tuple in $C_{\texttt{attempt1}}$, `ChkPassword` will restore the `attempts_left` and `password` variables and execution is restarted with input `"attempt1"` (listing 1 line 10).

Three, assume that the previous password was incorrect and the user enters `"attempt2"` for her second attempt. After storing the new cube $C_{\texttt{attempt2}}$ on disk, the system crashes *before* the incremented guard could be written to guarded memory (listing 4 line 4, t_2 in Fig. 4). This is an interesting point of failure as both cubes $C_{\texttt{attempt1}}$ as $C_{\texttt{attempt2}}$ can be considered fresh[6]. However, recovery based on either will preserve state continuity. This is obvious for cube $C_{\texttt{attempt2}}$ as this is the latest cube written to disk. Recovery from $C_{\texttt{attempt1}}$, however will purge any record of the login attempt made using `"attempt2"`. This is also safe as it was never used in any valuable computation (instructions after listing 1 line 24 were not executed yet). Hence, an attacker is not able to deduce any valuable information.

3.4.2 `libicen`: *State-Cont. Storage for n Modules*

By depending on scarce resources such as TPM NVRAM and guarded memory, `libice0` can in practice only provide state-continuous storage to a limited number of modules. `libicen` will alleviate this strain by using a single, unique `ice0` module to store freshness information on behalf of other modules. To safely exchange sensitive information between `libicen` and the `ice0` module, inter-module communication must guarantee endpoint authentication and confidentiality, integrity and freshness of messages. We will state this explicitly by passing a module identifier to `ice0` calls.

Creation of an initial state.

Similarly to `libice0`, an initial state of the module is stored by generating a new guard and cryptographic keys and writing a new cube to disk (see listing 6). Finally the

[6]The recovery step as displayed in listing 5, line 19, only accepts cube $C_{\texttt{attempt1}}$ as fresh. However, an attacker incrementing the guard stored in guarded memory, will trick `libice0` to accept cube $C_{\texttt{attempt2}}$ as being fresh as well.

```
void _init_step( State *state ) {
  mod.guard = gen_guard();
  mod.keys = gen_keys();
  hdd.write( new Cube( mod.guard, mod.keys,
       state ));
  mod.mode = Activated;
  ice0.store( mod.id, mod.keys, mod.guard);
}
```

Listing 6: `libicen`: Initialization of a new module

```
void _update_state( State *state ) {
  ++mod.guard;
  hdd.write( new Cube( mod.guard, mod.keys,
       state ));
  ice0.store( mod.id, mod.guard );
}
```

Listing 7: `libicen`: Updating a state

`ice0` module is requested to (state-continuously) store the keys and guard.

Updating a state.

To update a state, `libicen` first writes a new cube to disk, before the updated fresh guard is stored in `ice0` (see listing 7).

Recovering from a crash.

To recover from a crash, the (presumably) fresh cube is read from disk (see listing 8). Next, the keys and guard are requested from the `ice0` module. As the fresh guard is always stored safely in `ice0`, a cube with a correct MAC and that contains the fresh guard, must be fresh. Once the cube's freshness has been validated, `libicen` needs to generate a new guard, create and write a new cube to disk and store the new guard in `ice0` before a new step is taken.

The fact that a new guard is generated may be surprising

```
1  State *_recovery_step() {
     Cube cube = hdd.read();
3    ice0.retrieve( mod.id, &mod.keys, &mod.
         guard)
     if ( is_fresh( &cube ) ) {
5      State *state = extract(cube.state,
                              tpm.keys)
       mod.guard = gen_guard();
7      hdd.write( new Cube( mod.guard,
                            mod.keys,
9                           cube.state ) );
       mod.state = Activated;
11     ice0.store(mod.id, mod.keys, mod.guard)
       return state;
13   }
     else abort();
15 }
17
   bool is_fresh( Cube *cube ) {
19   return check_mac( cube, mod.keys ) &&
       mod.guard.value == cube.guard.value;
21 }
```

Listing 8: `libicen`: Recovering from a crash

since `libicen`'s guards never leak. But if this security measure is omitted, state continuity cannot be guaranteed. Let's reconsider `ChkPassword` and show that if no new guards are created upon recovery, an attacker can create (fresh) cubes for every password in a dictionary attack and later tests them one by one. To explain the first step, recall that modules are required to first store input-state tuples before processing input. This enables an attacker to input a password, store the resulting cube on disk and then crash the system *before* the input is committed; the system keeps crashing before writing the new guard to guarded memory (listing 7 line 4). These instructions are repeated for every password in the dictionary. In the second step of the attack, the module is finally allowed to check a password. If it is incorrect, the attacker crashes the system. Upon recovery the fresh cube is requested from disk, but as all cubes contain the *same* guard value, all are considered fresh and another guess can be made. This example shows that seemingly obvious state-continuous algorithms may be susceptible to subtle bugs and should be formally verified.

4. IMPLEMENTATIONS

To be feasible in practice, any system providing state-continuous storage needs to be (1) small enough to allow formal verification (2) operate seamlessly with legacy software (3) incur a low performance overhead and (4) not wear out TPM NVRAM. ICE is able to meet all these conditions, but depending on hardware support available, it is able to withstand different levels of hardware attacks. We describe implementations on platforms ranging from existing, commodity hardware platforms to distant future architectures.

4.1 ICE on Commodity Hardware

Given that SGX-enabled platforms will only become available in the near future, we implemented[7] a prototype of ICE on a commodity platform. A hypervisor-based protected-module architecture provided support for module isolation and we used CMOS memory as guarded memory. Obviously, since the hypervisor cannot prevent isolated modules to be evicted from the CPU cache to main memory in plaintext and CMOS memory is an easy attack vector, this implementation cannot protect against sophisticated hardware attacks.

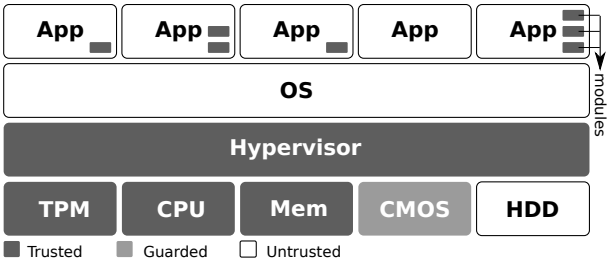

Figure 5: ICE provides state-continuity guarantees to isolated modules used by many state-of-the-art security architectures on commodity hardware

4.1.1 Architecture

We opted to implement our prototype on top of Fides [37], a pre-existing PMA architecture. Its support for secure communication between modules enables an elegant implementation of `libicen`, where `ice0` can be implemented as a protected module. Alternatively, other protected-module architectures could implement similar secure communication primitives, or include `ice0` as part of the security platform. Fides' architecture with trusted/untrusted components are displayed in Fig. 5.

While *any* non-volatile memory can serve as an alternative, CMOS memory is an interesting candidate for guarded memory. As it stores wall-clock time, it is updated every second and it must support a large number of write operations over its entire lifespan. Second, as it does not require a special communication protocol, it can be accessed easily and without much overhead. Being only accessible through direct I/O, it can also be isolated easily by hardware virtualization support.

4.1.2 Prototype Implementation

We added support to Fides for isolating and accessing CMOS memory. Using virtualization support to isolate programmed I/O, only 21 lines of code (LOC) had to be added to the hypervisor. Another 61 LOCs were required to implement system calls to access CMOS memory from the module. This totals the size of the hypervisor to 9,492 LOCs. While Fides at this moment does not support TPM chip accesses, we estimate, based on the Flicker [25] source code[8], that this straightforward effort would require an addition of less than 2,000 LOCs. As ICE only accesses the TPM at boot time, this does not impact performance.

While CMOS memory can be read/write accessed by the hypervisor, BIOS support for real-world implementations is required to allocate memory and exclude the area from its checksum to avoid that values written to CMOS memory are cleared on reboot. In practice, we must also ensure that the guard stored in CMOS memory is not lost when the system crashes while the previous guard is being overwritten. This

[7] Our research prototype is available at `https://distrinet.cs.kuleuven.be/software/sce/`

[8] `https://sparrow.ece.cmu.edu/group/flicker.html`

	libice0		libicen	
	asm	C	asm	C
ICE	0	372	0	341
SHA-512	0	371	0	371
AES-NI	1,566	176	1,566	176
Total	1,566	919	1,566	888

Table 1: Breakdown of `libice0` and `libicen`.

can be solved by implementing a two-phase commit protocol where first a new guard is written before overwriting the previous one. In the event of an unexpected crash, the old guard may not have been overwritten yet and both guards leak to an attacker. However, this does *not* impact security as such an event is similar to an unexpected loss of power before the guard is updated; An attacker can easily calculate the new guard based on the one leaked. This situation is covered by the formal, machine-checked proof.

As available space in CMOS memory is BIOS-specific, some systems may have insufficient space to store two hash values. There are two options (1) they may use an alternative hash function with a smaller hash value [6] or (2) only partially store the hash value (e.g. the 2 least significant bytes). As the value is only compared to the expected value, only the `operator<=` function of `libice0` needs to be trivially modified (listing 5, line 27). To prevent an attacker from guessing a correct value, the number of recovery attempts can be tracked by storing a counter in TPM NVRAM. This counter can be decremented before the provided guess is tested, preventing a crash attack. Full support for state continuity is in this case *not* required as the chances of repeated crashes during recovery are expected to be low in practice and thus losing a few recovery attempts is not an issue. To conservatively evaluate performance for a strong security implementation, we used the SHA-512 hash algorithm to create guard values. In Section 6 we will show that writing to CMOS is 5x more time consuming than `libice0`'s computations. Hence, writing less data to CMOS will have a positive impact on performance.

To implement `libice0` and `libicen`, we used the polarssl[9] library to calculate SHA-512 hash values and the Intel AES-NI reference implementation to take advantage of AES hardware support. This totals to 2,485 LOCs and 2,454 LOCs for `libice0` and `libicen` respectively (table 1).

4.2 SGX-Based Implementation

By relying on SGX' guarantees that enclaves never leave the CPU package in plaintext, ICE can also withstand physical attacks but some security measures must be taken. First, dedicated hardware support for guarded memory is required. As proposed in Section 3.1, dedicated registers can be added to the CPU storing the fresh guard while the system is powered on. When power is suddenly lost, the content of these registers can be backed up to non-volatile memory using a small capacitor. Non-volatile memory nor the capacitor are security sensitive. Only dedicated registers need to be protected from inspection by a (hardware) attacker but as they are included in the CPU package, they share the same protection as enclaves residing in one of the CPU's caches. To prevent that the contents of these dedicated registers can be accessed by a (software-level) attacker, we propose a simple permission mechanism; the first enclave that requested access is granted it exclusively. Attempts to access these registers from any other locations will be prevented.

Second, we must guarantee that the base guard stored in TPM NVRAM at the beginning of every execution stream, will not leak to an attacker. This can be established by setting up a secure channel from the `ice0` module to the TPM chip [44]. Authentication data can be sealed to the `ice0` module using SGX seal functionality [3, 17].

Third, in Section 3.4.2 we assumed that modules can easily and safely interact with one another. Fides supports such interaction explicitly by allowing modules to be authenticated and called from other modules. Unfortunately SGX enclaves cannot be called from other enclaves [17]. While this issue could be resolved with another hardware modification, it is not required. Using local attestation [3], enclaves can authenticate each other and set up a secure channel. As messages need to be passed in unprotected memory, they need to be confidentiality and integrity protected. A packet number must also be included to prevent replay attacks.

4.3 Distant Future Architectures

Given that TPM chips are already widely deployed, makes them a logical location to store freshness information and cryptographic keys over execution streams. Unfortunately, hardware attacks against the chip have been presented [36, 41, 47]. Moreover, the TPM chip is overly complex for our use case. This increases the risk of software vulnerabilities in the chip. We propose hardware replacements that provide stronger protection against physical attacks with only minimal hardware support.

To prevent ICE's cryptographic keys to leak to a hardware attacker launching offline attacks, we can replace it with a physically uncloneable function (PUF) [29]. PUFs are based on tiny variations in the manufacturing process of every individual hardware chip. This guarantees that PUFs are unique and are hard to copy. It has been shown that PUFs can be used to extract cryptographic keys. Their primary advantages over non-volatile memory, are (1) that they can only be read while they are powered and (2) physical tempering will destroy the PUF's intrinsic data. This makes them much more resilient against hardware attacks.

If we closely examine the `libice0` algorithm, we can observe that the base value of guards is always randomly selected. This enables it to be replaced by a reconfigurable PUF (rPUF) [20]. Similar to PUFs, rPUFs can be used to safely store random bits of data from an offline hardware attacker. But rPUFs have the additional functionality that they can be reconfigured; upon instruction the intrinsic secret data can be randomly modified. Once a rPUF has been reconfigured, it can never be reverted. Unfortunately, rPUFs still are a theoretical concept. Logical rPUFs have been proposed [18], where a PUF is combined with the contents of a non-volatile register. Only when the register contents is unchanged, will the logical rPUF return the same result. Upon reconfiguration, the register content is hashed and cannot be reverted. In contrast to "real" rPUFs, security sensitive information is still present after power-off and may be susceptible to attack.

While PUFs and rPUFs protect against a hardware attacker, we must ensure that malicious software on the platform cannot access these hardware primitives. This can be guaranteed by loading the `ice0` module on power-on [12] in

[9] `http://polarssl.org/`

memory and only allowing PUF/rPUF accesses from that memory region. In case the PUF/rPUF are implemented on a separate chip, a secure channel to the CPU needs to be established as well.

5. SECURITY EVALUATION

Since state-continuous algorithms must deal with sudden system crashes at any point in time, they are prone to subtle vulnerabilities. To guarantee ICE is indeed safe against a powerful attacker, we developed formal proofs of correctness.

5.1 Safety Properties

One of the properties that a safe state-continuous algorithm must guarantee, is that once a module starts computing with user-provided input, it must complete the initiated step or never advance at all. Provided input that was not yet used in any computation, however, may simply be discarded. To prove `libice0`'s safety, we first consider deterministic modules that only take their last state as input and later extend the proof to modules that also take non-deterministic (user) input. Finally we formally prove `libicen`'s security guarantees.

Safety of `libice0` in the event of deterministic modules was proven by a machine-checked proof[10] with the Coq proof assistant [5]. The proof required 118 definitions, 201 lemmas and totals 37,726 lines. For non-deterministic modules and `libicen`, we created formal proofs, but leave machine-checks as future work.

5.1.1 Model

We modeled the state of a concrete system as a 7-tuple (T, N, I, H, P, t, g) where record T holds the contents of the TPM's secure storage. To be able to model cryptographically secure random numbers, a monotonic counter is also kept in T. N keeps the content of the guarded non-volatile memory. Whether it can be accessed by an attacker depends on the mode of the countermeasure that is stored in the I record together with the current guard, cryptographic key and the current state of the module. H models the hard disk drive of the machine and stores the cubes. P models all public data, including cubes that were ever stored on disk and leaked guards from guarded memory. The algorithm itself is represented as a small program (see listing 9) that keeps advancing a module φ and is written as a term t. Finally a ghost element g tracks the last executed state and facilitates checking the proof.

Based on the state space $\mathcal{S}$ of these 7-tuples, we built a state machine with a step relation $S_M \subseteq \mathcal{S} \times \mathcal{S}$ where every step is either a program step or an attack step. A program step takes one evaluation step on term t. To ensure that our proof also holds in practice, we ensured that all instructions are feasible on commodity systems. For example, we do not consider that the TPM chip and guarded memory can be accessed simultaneously. We modeled the module requiring state continuity as a deterministic function φ. Whenever the module is executed with input state i, this argument is copied to ghost element g.

In an attack step an attacker is able to perform 4 operations:

[10]The correctness proof is publicly available at `https://distrinet.cs.kuleuven.be/software/sce/`

```
void ice_program( void ) {
 while ( true ) {
   switch (ice.mode, tpm.mode) {
     case (Activated, _):
       normal_step();
       break;
     case (Clear, Clear):
       init_step();
       break;
     case (Clear, Activated):
       recovery_step();
       break;
} } }
```

Listing 9: To prove `libice0`'s correctness, we created a small application that will keep advancing a module

- *Crash the system*: The system may crash at any point. This will (1) lift the protection of guarded memory and its contents becomes public knowledge. (2) I (i.e. `libice0`'s memory area) is cleared and does not leak. (3) `libice0` is restarted (term t is set to its initial value).
- *Modify HDD*: The contents of the hard disk drive may be modified by an attacker at any time. Cubes can be deleted, restored from public information in P, or cubes can be crafted by an attacker using publicly known cryptographic keys in P.
- *Modify guarded memory*: When the protection of guarded memory is down, an attacker is able to modify its contents and set it to any publicly derivable guard value. A guard $g' = (v', i')$ is publicly derivable from guard $g = (v, i)$ when its guard value v' can be computed from v:

  $$\exists n \in \mathbb{N}, v' = \mathrm{Hash}^n(v)$$

  The guard index is not considered and can be chosen arbitrarily.
- *Use random number*: At any time an attacker is able to request a new random number from the TPM chip.

5.1.2 `libice0`'s State-Continuity Guarantees

To prove `libice0`'s correctness, we differentiate between modules that take non-deterministic input and modules that only operate on their last state.

`libice0`: State Continuity of Deterministic Modules.

Before we discuss the proof in more detail, we first introduce some definitions. We define R^* as the reflexive-transitive closure of a state relation R i.e. $R^* = \cup_{n \in \mathbb{N}} R^n$. The image $R(X)$ of a set X under a relation R is defined as $R(X) = \{s' | \exists s \in X.(s, s') \in R\}$. We also define the composition $P; R$ of a state predicate P and a state relation R as $P; R = (P \times \mathcal{S}) \cap R$. Similarly, we define $R; P$ as $R; P = R \cap (\mathcal{S} \times P)$

To prove state continuity we use rely-guarantee reasoning and reason about reset, interference and program steps separately. We say a submachine with step relation S is *safe* under a precondition P, a rely condition R, and a guarantee condition G, denoted safe(P, R, G, S), if, when starting

from a state that satisfies the precondition, all steps by the submachine satisfy the guarantee condition, assuming that all steps by the environment satisfy the rely condition:

$$\text{safe}(P, R, G, S) = (S \cup R)^*(P); S \subseteq G$$

Finally we define S_M as a step relation $S_M \subseteq \mathcal{S} \times \mathcal{S}$ where each step is either a program step or an attacker step.

THEOREM 1 (`libice0`'s DETERMINISTIC SAFETY). *We wish to prove the following:*

$$safe(\{s_0\}, Id, A, S_M)$$

where the precondition allows just the initial state s_0, the rely condition is the identity relation (since there is no environment), and the guarantee condition is the set of allowed steps. A step is allowed when `libice0` *either calls φ with its last output or it is a stutter step where φ is not called or it re-executes with the last input:*

$$A = \{(s, s') \in S_M | g(s') = g(s) \vee g(s') = \varphi(g(s))\}$$

First we separate reset steps from non-reset steps: $S_M = S_{\text{Reset}} \cup S_{\text{Nonreset}}$. We can do so using the following inference rule:

$$\text{Reset}\frac{\begin{array}{c} S = S_{\text{Reset}} \cup S_{\text{Nonreset}} \qquad I_{\text{Reset}}; R \subseteq R; I_{\text{Reset}} \\ \text{safe}(Q_{\text{Reset}}, R, (G; I_{\text{Reset}}), S_{\text{Nonreset}}) \qquad P \subseteq Q_{\text{Reset}} \\ I_{\text{Reset}}; S_{Reset} \subseteq G; Q_{\text{Reset}} \qquad Q_{\text{Reset}} \subseteq I_{\text{Reset}} \end{array}}{\text{safe}(P, R, G, S)}$$

Here, the reset postcondition Q_{Reset} is a state predicate that always holds immediately after a reset. It must also hold initially ($P \subseteq Q_{Reset}$). The reset invariant I_{Reset} is a state predicate that holds in every reachable state, i.e. it is preserved by the reset steps as well as the non-reset steps. It follows that a reset step always starts in a state that satisfies the reset invariant. The reset postcondition must imply the reset invariant.

We can further categorize the non-reset steps S_{Nonreset} into program steps S_{Prog} and interference steps S_{Itf}. During an interference step, an attacker may, for example, modify the contents of the hard disk, or of guarded memory when its protection is not enabled. Again, we wish to reason about these steps separately. We can do so using the following inference rule:

$$\text{Interference}\frac{\begin{array}{c} S = S_{\text{Prog}} \cup S_{\text{Itf}} \qquad I; S_{\text{Itf}} \subseteq G; I \cap G_{\text{Itf}} \\ I; R \subseteq R; I \qquad P \subseteq I \\ \text{safe}(P, R \cup G_{\text{Itf}}, G; I, S_{\text{Prog}}) \end{array}}{\text{safe}(P, R, G, S)}$$

Here, a global invariant I is established by the precondition P and maintained by all steps. Furthermore, interference steps satisfy an interference guarantee G_{itf}. Program steps are verified under a rely that is the union of the global rely and the interference guarantee.

Now that we have isolated the program steps, we wish to perform simple forward reasoning to verify these. For this purpose, we define the following auxiliary safety judgment:

$$\begin{aligned} \text{safe'}(s, R, G, S)_0 &= \text{true} \\ \text{safe'}(s, R, G, S)_{n+1} &= \forall s' \in R(s). \forall s'' \in S(s'). \\ &\qquad (s', s'') \in G \wedge \text{safe'}_n(s'', R, G, S) \end{aligned}$$

We have the following inference rule:

$$\text{Program}\frac{\begin{array}{c} Id \subseteq R' \qquad R \subseteq R' \qquad R'; R' \subseteq R' \\ \forall n \in \mathbb{N}, s \in P \;.\; \text{safe'}_n(s, R', G, S) \end{array}}{\text{safe}(P, R, G, S)}$$

Note that safe' assumes that the environment performs a single R' step before every program step. Therefore, R' must subsume the reflexive-transitive closure of R. This is expressed by the first three premises of the inference rule.

If the program contains a loop, such as the `while`-loop in `libice0`'s main function (see listing 9), we can verify it as follows:

$$\text{Loop}\frac{\begin{array}{c} s \in I \\ \forall m \in \mathbb{N}, (\forall k < m, s' \in I \;.\; \text{safe}'_k(s', R, G, S)) \\ \Rightarrow \forall s' \in I \;.\; \text{safe}'_m(s', R, G, S) \end{array}}{\text{safe'}_n(s, R, G, S)}$$

Using loop invariant I we prove that, if the program is safe starting from I for less than m steps, then it is safe starting from I for m steps. This is a classical inductive proof.

The invariants that we had to come up with were reasonably big and contained a lot of information relating to how we modeled the secure random number generator. For example, we had to prove for every step that guards that were leaked all were created using the TPM's SRNG. This ensured that newly generated guards were not yet in the public domain. More interesting was the cube invariant stating that any cube in the public domain and that is seen by the algorithm as fresh, contains as module state either the ghost state g or the result of $\varphi(g)$. This led to a case split of the normal step, where a new state is stored that depends on the previous round of `libice0`'s main function (see listing 9) . Either the algorithm was initialized or took a normal step in the previous round, or has recovered from a crash. In the former case, fresh public cubes contain the last state (and equal the `libice0`'s internal memory) that was given to φ and the algorithm will advance the module to a new state. Alternatively, the last round was a recovery step in which case `libice0` may execute a stutter step, depending on when exactly the system was reset: before or after a new cube was written to disk and the step committed by writing the successor guard to guarded memory. In case the cube was not yet stored, fresh cubes contain the same state as is stored in the ghost state g. Also note that when the system was reset after the cube was written to disk but before the step was committed, both options are feasible since the contents of guarded memory is now publicly accessible and an attacker could complete the step.

`libice0`: *State-Continuity of Non-Deterministic Modules.*

In the previous paragraph we proved state continuity for deterministic modules. Using an alternative formulation, we proved that when an application step φ is taken from application state a in machine state s and followed by any number of attack steps, then for the next application step it follows that either a new step was taken ($a' = \varphi(a)$) or the module re-executed the last step ($a' = a$):

$$s \xrightarrow{\varphi(a)} s' \xrightarrow{\text{att}}{}^* s'' \xrightarrow{\varphi(a')} s''' \Rightarrow a' = a \vee a' = \varphi(a)$$

Based on this theorem we can easily extend our model to allow non-deterministic inputs to the module. Given a

module that operates on its previous state and (user) input, we prove that once it commits to an input, it will either use that input to advance its state or it will never advance (e.g., because the attacker crashes the system and erases the fresh cube).

THEOREM 2 (`libice0`'S NON-DETERMINISTIC SAFETY). *More formally, consider a module φ that accepts input i on application state a and machine state s followed by any number of attack steps. When the module takes another step, it will either advance to the next state, or it will re-execute its previous step with the same input:*

$$s \xrightarrow{\varphi(a,i)} s' \xrightarrow{att}{}^{*} s'' \xrightarrow{\varphi(a',i')} s''' \\ \Rightarrow a' = a \wedge i' = i \vee a' = \varphi(a,i) \qquad (i' \in I)$$

where I is the set of possible input values.

The core principle of `libice0`'s deterministic state-continuity proof is that we know that a state update is committed when we successfully incremented the guard in guarded memory. At that point we can feed module φ other input. We use this knowledge to partition the non-deterministic module φ as $\varphi = \varphi_i \circ \varphi_c$ where φ_i requests input from the user and φ_c deterministically computes a new state with the input state. By requesting the user's input after a computation step and storing it in a new state, only two possibilities arise. One possibility is that the system crashes after the result of φ_i was committed. In that case the input is stored and will be provided to φ in the next invocation where φ_c first computes the result. The other possibility is that the state was not committed. In that case φ_c will recompute the last state and request the user again for input. This does *not* violate state continuity since the input value is not used in any computation of the module. Hence, no information about the result of φ_c can leak to an attacker.

5.1.3 `libicen`'s State-Continuity Guarantees

While `libice0` only provides state-continuity properties for a single module, `libicen` supports a virtually unlimited number of modules. As we did for `libice0` we will focus on security guarantees for deterministic modules. Support for non-determinism can be provided in a similar fashion as was explained in the previous paragraph.

THEOREM 3 (`libicen`'S SAFETY). *Given a deterministic module φ, we will argue that* `libicen` *ensures state continuity: φ will only re-execute the last step or take a new step. More formally we wish to prove:*

$$safe(\{s_0\}, Id, A, S'_M)$$

Where we define S'_M as a step relation $S'_M \subseteq \mathcal{S} \times \mathcal{S}$ where each step is either a program step of `libicen` *or an attacker step. For s_0, Id and A, we use the same definitions as earlier.*

In order to prove `libicen`'s correctness, we could take the same approach as we did for `libice0`. Given the close similarity between `libice0` and `libicen`, however, we could also use the proof of `libice0` and reason that the modifications of `libicen` do not affect state continuity. We will take this approach using three consecutive transformation steps α, β, γ and prove that each transformation preserves state continuity:

$$\mathcal{S} \xrightarrow{\alpha} \mathcal{S}_1 \xrightarrow{\beta} \mathcal{S}_2 \xrightarrow{\gamma} \mathcal{S}_n$$

First consider transformation α that transforms a state $\mathcal{S}$ of `libice0` into a state $\mathcal{S}_1$ where TPM NVRAM is used instead of guarded memory. It is obvious that this does not affect the security guarantees of `libice0` since contents of this memory will never leak to an attacker. Thus the attacker grows weaker instead of stronger.

After the α transformation, two guards are stored in the TPM's NVRAM: the base guard $(n, 0)$ and a guard $(\text{Hash}^i(n), i)$ used to determine which cube is fresh. Since neither of them can be modified by an attacker and the latter is always a successor of the former, the base guard can be omitted. The second transformation β will remove the base guard and modify the code of the recovery step. Since all changes happen within the module's memory area which cannot be influenced by an attacker in any new way, modifications of these instructions do not affect state continuity.

Finally consider an abstraction function γ that abstracts states $\mathcal{S}_2$ in states $\mathcal{S}_n$ where all interactions with the TPM's NVRAM are replaced with a call to an `libice0` module. Hence, we can rely on `libice0` to ensure integrity, confidentiality and state continuity of the stored data. It is also obvious that combining multiple steps into a single atomic step does not affect state continuity.

5.2 Liveness Properties

Given our strong attacker model, it is infeasible that *any* state-continuity algorithm on commodity hardware is able to guarantee that a module is always able to advance. An attacker could, for example, always delete the fresh cube. Such situations could be resolved in ICE by re-executing the initial step, at the cost of losing all previously stored states.

We do however wish to guarantee that progress can always be made in the event of benign events, such as a sudden loss of power during any step in the execution of the algorithm.

`libice0`'s liveness properties.

To guarantee liveness under benign events, `libice0` needs to be able to recover from a crash during every step of its execution. An important distinction can be made based on the value of the `tpm.mode` field. This value indicates whether the algorithm has been initialized correctly. A crash before this value is set, will result in a re-execution of the initialization step. After setting this value, all crashes will result in the execution of the recovery step. To ensure that the initialization step may be re-executed in the event of a sudden crash, the `tpm.mode` value is set last.

After initialization we may update the state or we have to recover the fresh state. In the former case we make sure to first store the cube before we update the content of guarded memory. Recovery of a state is more challenging since we have to modify the guards in both guarded and TPM NVRAM memory. After creating a new cube with the module's fresh state and storing it on disk, we enable protection of guarded memory and write the new guard to it before we update TPM NVRAM memory. This has an important consequence: in case the system crashes before the recovery step is completed, the old guard may already have been overwritten. This would prevent the re-execution

of the recovery step. Therefore we require that a backup of this guard is stored on disk *before* the recovery step is called.

`libicen`*'s liveness properties.*

Ensuring liveness of `libicen` is straightforward since we only have to deal with two non-volatile data objects: cubes and calls to `libice0`. For obvious reasons we ensure to first store new cubes on disk. The `libice0` algorithm guarantees that its modifications are done atomically and are always retrievable.

6. PERFORMANCE EVALUATION

In this section we evaluate the performance of our prototype implementation. To compare the performance impact of a solid state drive (SSD) against a rotating hard drive (HDD), we used two machines with comparable hardware. The first machine, a Dell Latitude E6510, a mid-end consumer laptop, is equipped with an Intel Core i5 560M processor running at 2.67 GHz and 4 GiB of RAM. It is also equipped with a magnetic hard disk (HDD), a Broadcom TPMv1.2 chip and CMOS memory. The second testing laptop is a Dell Latitude E6520, has an Intel Core i5-2520M CPU running at 2.50GHz and is equipped with an SSD.

Hardware Benchmarks.

To better understand the performance cost of ICE compared to TPM operations, we performed 4 benchmarks on the Latitude E6510: read/write accessing TPM NVRAM, extending PCR registers and generating random numbers. To perform these tests, we developed small TPM applications using the TrouSerS[11] open-source software stack. We also modified the tpm_tis driver to keep timing measurements. Each test was run 100 times and transferred 128 bytes to/from the TPM. Figure 6 displays the median time for each test graphically. All operations take a significant amount of time to complete. Especially writing to TPM NVRAM takes 4x longer than reading from it. Related work shows similar results for TPM chips from other vendors [30].

We also performed a similar benchmark on CMOS memory. We performed 10,000 one-byte write operations and measured the time using the `rdtscp` instruction. Writing to CMOS takes about 3μs/byte, significantly faster than writing to TPM NVRAM. We attribute this difference to the fact that CMOS memory is connected to the SPI-bus [16] and does not require a heavy communication protocol as does the LPC-connected TPM chip.

Finally, we measured the median time of writing 10,000 128 bytes files to both HDD and SSD disks. As Figure 6 shows, accessing the SSD disk is 5.4 times faster than writing to TPM NVRAM. Writing to a magnetic disk is more costly.

Microbenchmarks.

To measure the performance of both `libice0` and `libicen` libraries, we implemented two modules. The first module implements a password verification function and limits the number of attempts that can be made before the user is locked out indefinitely. The benchmark provided this module with 10,000 wrong password guesses and measured the median time per guess. Measurements show (see Table 2) that for a single step only 0.06ms (0.43%) were

[11] `http://trousers.sourceforge.net/`

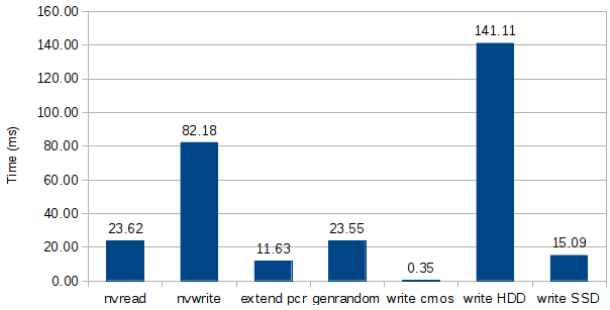

Figure 6: Microbenchmarks of various TPM operations show a significant difference in performance cost of CMOS and disk accesses. Where applicable, 128 bytes were transferred.

spent on computation when the module was linked with the `libice0` library. When we used `libicen`'s services, two cubes need to be created and computation time increased to 0.13ms (0.71%). To securely write guards to CMOS memory, 0.33ms were spent (2.17% and 1.82% for `libice0` and `libicen` resp.). This shows a much higher cost to write guards to CMOS compared to calculation time. But most of the time was spent committing cubes to solid state disk (97.40% and 97.47% for `libice0` and `libicen` resp.). `libicen` does *not* spend twice the amount of time writing cubes to disk. Cubes only need to be committed before a guard is incremented. Hence, `libicen`'s cubes can be stored temporarily in memory and transferred to disk together with `ice0`'s new cube *without* modifying the algorithm (see listing 7, lines 3-4), reducing disk access times.

While most TPM chips NVRAM area is limited to 1,280 bytes [30], it could be used to provide (state-continuous) storage to a single module to avoid disk overhead. To show that such a module would still benefit from ICE, we implemented a second benchmark called Noop. It does not perform any computation but only stores a state of 1,280 bytes. As expected given the performance of SHA-512 and Intel's AES hardware support, the increase in computation cost is negligibly with only 0.01ms. As cubes are still smaller than disk sectors, costs of disk accesses are comparable to the Password benchmark. This totals the cost of storing new data in Noop at 15.05ms to 17.65ms for `libice0` and `libicen` resp; significantly faster than 82.18ms to access TPM NVRAM. Finally we performed these tests on the Latitude E6510 which is equipped with a magnetic HDD. As expected, the cost of writing cubes to disk increased significantly and now accounts for 99.63%-99.74%. For both benchmarks `libicen` consistently takes more time writing cubes to disk than `libice0`. We attribute this behavior to the way we implemented its write function: merging `ice0`'s and `libicen`'s cubes takes us 3 `write` system calls before system buffers are flushed.

Expected Impact of Dedicated Hardware.

These benchmarks show that only up to 0.14% of time is spent on computation. With dedicated hardware performance can be increased significantly.

Writing guards to CMOS memory is about 2.4 times more costly than computation and takes up to 0.31% of the time in case of a revolving HDD and up to 2.17% on our SSD

	Password		Noop	
SSD (in ms)	`-lice0`	`-licen`	`-lice0`	`-licen`
computation	0.06	0.13	0.07	0.14
writing guard	0.33	0.33	0.33	0.33
writing cubes	14.61	17.42	14.65	17.19
total	15.00	17.87	15.05	17.65

HDD (in ms)	`-lice0`	`-licen`	`-lice0`	`-licen`
computation	0.06	0.12	0.07	0.13
writing guard	0.35	0.35	0.35	0.35
writing cubes	112.80	183.23	111.54	183.83
total	113.21	183.71	111.96	184.31

Table 2: Microbenchmarks for `libice0` and `libicen`

testing platform. Hardware support for guarded memory, as described in detail in Section 3.1, would reduce overhead of this operation to almost zero.

But committing cubes to disk forms the real bottleneck, requiring up to 97.47% (for SSD) to 99.74% (for HDD) of the time. Recently Viking Technology [43] and Micron Technology [42] announced that they will ship capacitor-backed RAM to market. Operating similar to guarded memory, these hardware components contain fast, volatile memory that is written to flash memory when power is suddenly lost. Adding these hardware components to our system would eliminate disk access completely.

In summary, benchmarks show that our prototype implementation on commodity hardware already outperforms TPM NVRAM write operations by almost 5 times. Adding dedicated hardware support for guarded memory and capacitor-backed RAM, may even enable state updates 587 times faster than TPM NVRAM accesses!

7. IMPLICATIONS TOWARDS HARDWARE SECURITY MODULES

The TPM chip does not allow software modules to be executed within its protected boundaries. Instead, it is shipped with all supported security primitives stored in dedicated memory. This results in a number of drawbacks. First, as more resources are required, power usage and hardware costs increase. To minimize the costs of the PC platform, the TPM is equipped with a slow operating processor and limited NVRAM. Other specifications have been developed for various platforms such as the Mobile Trusted Module (MTM) for mobile devices. Unfortunately power and economic constraints still form an obstacle for low-end applications such as sensor networks. Second, it increases the possibility of software vulnerabilities in the TPM chip. Some vendors already struggle with its complexity [32] and as functionality is added, the specification may become even more complex.

Instead of using a separate chip, recent research shows that strong security guarantees can also be provided when modules execute on the same processor as untrusted, legacy software. Agten et al. [2] describe a fully abstract compilation scheme and proof that modules at machine code level can only be attacked *iff* vulnerabilities also exist at source code level. Furthermore, Noorman et al. [27] apply minimal hardware modifications to implement a protected-module architecture on a low-end processor.

ICE shows that with little additional hardware support, various security primitives can be implemented as protected modules. This has several benefits: (1) By executing protected modules on the same processor as legacy software, overall power consumption and economic cost is reduced, enabling security measures to be ported to low-end devices (2) Software primitives can be added and updated after the chip was manufactured or deployed. (3) Hardware optimizations will increase performance of the overall chip, protected modules and untrusted software alike. (4) The overall complexity of the system is reduced. We believe that these advantages can affect future versions or revisions of hardware security modules, such as the TPM chip and the security of low-end devices in general.

8. RELATED WORK

Most research prototypes do not consider state continuity, leaving them vulnerable to attack. Others propose special-purpose solutions without addressing resource constraints. We divide these research results in four categories: results proposing hardware modifications, results that isolate persistent storage, module-isolation architectures that only require a minimum TCB and special-purpose applications.

Hardware Modifications.

XOM [23] protects against an attacker that is able to snoop buses and modify memory by encrypting data and code before it is sent to memory. While it makes it significantly more difficult to successfully attack the system, Suh et al. [39] argue correctly that it is vulnerable to a memory replay attack where stale memory pages are returned to the processor. Their Aegis architecture mitigates this replay attack by storing hash trees of memory pages in a secure location. When a memory page is loaded into the processor's cache, its freshness is checked by recalculating and comparing the hash values. Subsequent research results also defend against replay attacks [8, 17, 46].

Memory replay attacks differ from rollback attacks in that memory contents is replayed *while* the system is up and running. This enables much easier security measures.

Schellekens et al. [34] propose an embedded-systems architecture to store a trusted module's persistent state in invasive-attack-resistent, non-volatile memory. Their solution implements a light-weight authenticated channel between the trusted module and non-volatile memory. Freshness of the stored data is guaranteed per read/write instruction and based on a monotonic counter. As their approach assumes that write instructions to non-volatile memory and increments of the monotonic counter are atomic, unexpected loss of power enables a rollback attack. We believe that their approach can be fixed by keeping a log of instructions in secure non-volatile memory that need to be completed in case power suddenly fails. On higher-end systems however, only the TPM NVRAM can be used for such purposes and would lead to significant performance overhead. ICE eliminates the TPM chip on the performance critical path altogether.

Research Systems Isolating Persistent Storage.

Many architectures rely on a large TCB that includes isolation of persistent storage [13, 35, 40]. In such cases protection against rollback attacks are trivial: modules/programs can overwrite their state on disk. In practice however, software vulnerabilities in their TCB may be exploited

and state-continuity support is hard to guarantee. These systems are also not able to defend against disk clone attacks. In contrast, ICE provides strong guarantees while only relying on a very limited TCB.

Protected-Module Architectures.

In recent years many security architectures have been proposed that attempt to minimize the TCB by only providing strong isolation guarantees of modules [4, 17, 24, 25, 33, 37]. Persistent storage can only be accessed via services provided by the untrusted legacy operating system. None of them address the issue of state continuity.

Many of these systems can be adapted to use the state-continuity approach presented by Parno et al. [30]. This seminal work called Memoir, is to the best of our knowledge the first and only work that addresses the issue of state continuity in protected-module architectures. Based on Flicker [25], Memoir uses TPM NVRAM to store freshness information upon every state update. This significantly limits the applicability of their solution as NVRAM is slow and only required to support up to 100K writes. The authors acknowledge this constrained and propose two solutions: (1) adding capacitor-backed RAM to the TPM chip and (2) Memoir-Opt, an alternative approach that stores freshness information in (volatile) TPM PCR registers that are written to NVRAM when power is lost unexpectedly. Both solutions rely on an uninterruptible power source to safely store freshness when power suddenly fails. Failure in this mechanism can lead to a rollback attack. ICE, in contrast, is a passive state-continuity system that does not rely on an uninterruptible power source to guarantee security; detaching the capacitor would only prevent stateful modules from recovering their state but states could not be rolled back. Moreover, in ICE the speed of updates to state-continuous modules is only limited by the processor and (untrusted) non-volatile memory, not by the TPM chip.

Special-Purpose Applications.

Chun et al. proposed the creation of append-only memory [10] to harden existing distributed algorithms to defend against equivocation: making different statements to different nodes in the network. An implementation with a minimal TCB was left as future work.

Levin et al. propose TrInc [21], a specialized system to attest successive monotonic counters, to achieve similar results. TrInc assumes a dedicated device that is able to locally store attestation requests of monotonic counters. In the event of an unexpected loss of power, clients can request the last signed attestations. This approach is similar to solutions were disk space is isolated, but incurs only a limited TCB. ICE provides a more generic, low-overhead alternative with only limited hardware modifications.

More recently Kotla et al. proposed a system [19] that allows offline data access while guaranteeing that (1) a user cannot deny offline accesses without failing an audit and (2) after proving that a user did not access the data, it cannot be accessed in the future. While their solution is interesting and does not require any software to be trusted, it only solves state-continuity in this specific setting.

9. CONCLUSION

Providing support for state continuity is challenging as including non-volatile memory on-chip requires modification of fabrication processes. But off-chip storage of freshness information can be slow (e.g. TPM NVRAM) or vulnerable to attack. We presented ICE, a state-continuous system and algorithm with two important properties: (1) only at boot time is the (slow) TPM chip accessed. State updates after the system booted only require updates to dedicated registers backed off-chip by a capacitor and non-volatile memory. (2) ICE is a passive security measure. An attacker interrupting the main power supply or any other source of power, cannot break state-continuity. We believe that the importance of ICE lies in the fact that it shows that with only limited and cheap hardware support, it enables the development of software-only implementations of trusted computing primitives. This presents an interesting direction for future versions or revisions of hardware security modules (e.g., the TPM chip) and may provide an interesting approach to increase security in low-end, resource-constrained applications such as sensor networks.

Acknowledgments

The authors thank all reviewers and proofreaders of the paper for their useful comments. We also thank Frédéric Vogels and Dominique Devriese for their help with Coq.

This work has been supported in part by the Intel Lab's University Research Office. This research is also partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. Raoul Strackx holds a PhD grant from the Agency for Innovation by Science and Technology in Flanders (IWT).

10. REFERENCES

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Operating Systems Review*. ACM, 2005.

[2] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *2012 IEEE 25th Computer Security Foundations Symposium (CSF 2012)*, pages 171–185, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[3] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP*, volume 13, 2013.

[4] A. Azab, P. Ning, and X. Zhang. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 375–388. ACM, 2011.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[6] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varici, and I. Verbauwhede. Spongent: a lightweight hash function. In *Proceedings of the 13th international conference on Cryptographic hardware and embedded systems*, CHES'11, pages 312–325, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[8] D. Champagne and R. Lee. Scalable architectural support for trusted software. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[9] E. M. Chan, J. C. Carlyle, F. M. David, R. Farivar, and R. H. Campbell. BootJacker: Compromising computers using forced restarts. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 555–564, New York, NY, USA, 2008. ACM.

[10] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Operating Systems Review (OSR'07)*, volume 41, pages 189–204. ACM, 2007.

[11] D. Dolev and A. C. Yao. On the security of public key protocols. volume 29, pages 198–208, 1983.

[12] K. El Defrawy, D. Aurélien Francillon, and G. Tsudik. Smart: Secure and minimal architecture for (establishing a dynamic) root of trust. In *Proceedings of the Network & Distributed System Security Symposium (NDSS), San Diego, CA*, 2012.

[13] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Operating Systems Review (OSR'03)*, volume 37, pages 193–206. ACM, 2003.

[14] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, pages 45–60, 2008.

[15] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 11. ACM, 2013.

[16] Intel Corporation. *Intel 6 Series Chipset and Intel C200 Series Chipset*, 2011.

[17] Intel Corporation. *Software Guard Extensions Programming Reference*, 2013.

[18] S. Katzenbeisser, U. KoÃğabas, V. Leest, A.-R. Sadeghi, G.-J. Schrijen, H. Schröder, and C. Wachsmann. Recyclable PUFs: Logically reconfigurable PUFs. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems (CHES 2011)*, volume 6917 of *Lecture Notes in Computer Science*, pages 374–389. Springer Berlin Heidelberg, 2011.

[19] R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester. Pasture: secure offline data access using commodity trusted hardware. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12)*, pages 321–334. USENIX Association, 2012.

[20] K. Kursawe, A.-R. Sadeghi, D. Schellekens, B. Skoric, and P. Tuyls. Reconfigurable physical unclonable functions-enabling technology for tamper-resistant storage. In *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on*, pages 22–29. IEEE, 2009.

[21] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *NSDI'09*, volume 9, pages 1–14, 2009.

[22] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, volume 4, pages 9–9, 2004.

[23] D. Lie, T. Chandramohan, M. Mark, L. Patrick, B. Dan, M. John, and H. Mark. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, volume 35, pages 168–177. ACM, 2000.

[24] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'10)*, May 2010.

[25] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, pages 315–328. ACM, Apr. 2008.

[26] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 10. ACM, 2013.

[27] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium (Usenix'13)*. USENIX Association, Aug. 2013.

[28] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*, pages 1–20. Springer-Verlag, 2006.

[29] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld. Physical one-way functions. volume 297, pages 2026–2030. American Association for the Advancement of Science, 2002.

[30] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'11)*, May 2011.

[31] M. Patrignani, D. Clarke, and F. Piessens. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13)*, volume 8301 of *LNCS*, pages 176–191, 2013.

[32] A.-R. Sadeghi, M. Selhorst, C. Stüble, C. Wachsmann, and M. Winandy. TCG inside?: a note on TPM specification compliance. In *Scalable trusted*

computing, pages 47–56. ACM, 2006.

[33] R. Sahita, U. Warrier, and P. Dewan. Protecting Critical Applications on Mobile Platforms. volume 13, pages 16–35. Intel, 2009.

[34] D. Schellekens, P. Tuyls, and B. Preneel. Embedded trusted computing with authenticated non-volatile memory. In P. Lipp, A.-R. Sadeghi, and K.-M. Koch, editors, *First International Conference on Trusted Computing and Trust in Information Technologies (TRUST'08)*, Lecture Notes in Computer Science, pages 60–74. Springer Berlin Heidelberg, 2008.

[35] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'06)*, pages 161–174, New York, NY, USA, 2006. ACM.

[36] E. R. Sparks. A security assessment of trusted platform modules. Technical report, Technical report, Department of Computer Science Dartmouth College, 2007.

[37] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS'12)*, October 2012.

[38] R. Strackx, F. Piessens, and B. Preneel. Efficient Isolation of Trusted Subsystems in Embedded Systems. pages 344–361. Springer, 2010.

[39] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing (ICS'03)*, ICS '03, pages 160–171, New York, NY, USA, 2003. ACM.

[40] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI'06)*, pages 279–292. USENIX Association, 2006.

[41] C. Tarnovsky. Deconstructing a "secure" processor. volume 2010, 2010.

[42] M. Technology. Hybrid memory - bridging the gap between dram speed and nand nonvolatility. Technical report.

[43] V. Technology. NV-DIMM: Achieving greater ROI from SSDs. Technical report.

[44] Trusted Computing Group. Design Principles Specification Version 1.2. 2011.

[45] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 430–444, Washington, DC, USA, 2013. IEEE Computer Society.

[46] P. Williams and R. Boivie. CPU support for secure executables. pages 172–187. Springer, 2011.

[47] J. Winter and K. Dietrich. A hijacker's guide to the LPC bus. In *Proceedings of the 8th European conference on Public Key Infrastructures, Services, and Applications (EuroPKI'11)*, pages 176–193. Springer, 2012.