Ariadne: A Minimal Approach to State Continuity

Raoul Strackx and Frank Piessens, *Katholieke Universiteit Leuven*

**This paper is included in the Proceedings of the
25th USENIX Security Symposium**

August 10–12, 2016 • Austin, TX

Ariadne: A Minimal Approach to State Continuity

Raoul Strackx
iMinds-DistriNet, KU Leuven,
3001 Leuven, Belgium
raoul.strackx@cs.kuleuven.be

Frank Piessens
iMinds-DistriNet, KU Leuven,
3001 Leuven, Belgium
frank.piessens@cs.kuleuven.be

Abstract

Protected-module architectures such as Intel SGX provide strong isolation guarantees to sensitive parts of applications while the system is up and running. Unfortunately systems in practice crash, go down for reboots or lose power at unexpected moments in time. To deal with such events, additional security measures need to be taken to guarantee that stateful modules will either recover their state from the last stored state, or fail-stop on detection of tampering with that state. More specifically, protected-module architectures need to provide a security primitive that guarantees that (1) attackers cannot present a stale state as being fresh (i.e. rollback protection), (2) once a module accepted a specific input, it will continue execution on that input or never advance, and (3) an unexpected loss of power must never leave the system in a state from which it can never resume execution (i.e. liveness guarantee).

We propose Ariadne, a solution to the state-continuity problem that achieves the theoretical lower limit of requiring only a single bit flip of non-volatile memory per state update. Ariadne can be easily adapted to the platform at hand. In low-end devices where non-volatile memory may wear out quickly and the bill of materials (BOM) needs to be minimized, Ariadne can take optimal use of non-volatile memory. On SGX-enabled processors, Ariadne can be readily deployed to protect stateful modules (e.g., as used by Haven and VC³).

1 Introduction

Computing devices have become ever more diverse, ranging from cloud computing platforms and super computers to embedded systems used in Internet of Things (IoT) applications. The familiar multi-level approach to security where a more privileged layer has full control over software running on top, is ill-suited for many of these applications; clients of cloud providers may fear rogue employees or government subpoenas targeting their provider that may reside in a different country [7]. On embedded devices paging and privilege layers may be too power/energy expensive to be applied.

Protected-module architectures (PMAs) take another, non-hierarchical approach. After almost a decade of research [5,6,8,9,11–13,22,27,28,30,31,41,45,48,49], two key primitives have emerged: isolation and key derivation. The *isolation* mechanism ensures that a protected module is completely isolated from any other piece of code running on the system, including other protected modules. Only when the instruction pointer points to a memory location within the module, can a module's memory regions be accessed. All other attempts from different locations are blocked by the architecture. Only a module's entry points are an exception and can be accessed from any location. Once an entry point is called, the instruction pointer points within the module's memory region and it's secrets can be accessed.

A *key derivation mechanism* provides a unique, unforgeable key for each protected module. It is usually derived from a platform specific key – that can only be accessed directly by the platform itself – and the measurement of the module when it was created. This implies that only identical modules can derive the same cryptographic key. An attacker modifying the module before it was properly isolated, will cause a variation in the module's measurement and eventually in the cryptographic key that was derived. This makes it ideal to seal data to a specific module. Data can be integrity and confidentiality protected by the derived key, and stored on disk. As only the identical module can derive the same key, the stored sensitive data cannot be accessed by an attacker.

Related work showed that these minimal requirements are small enough to be implemented directly in hardware [31], even for embedded devices [8, 12, 22, 30]. With the arrival of Intel Skylake CPUs in August 2015, equipped with Intel Software Guard eXtensions (SGX) [3, 18, 29], PMAs are now available on commod-

ity devices.

In parallel, research was conducted on how the security properties provided by PMAs can be leveraged to provide provable security guarantees. Agten et al. [2] and others in subsequent work [1, 33, 34] showed that by adding limited security checks at runtime, fully-abstract compilation can be guaranteed; all security properties that hold at source-code level, can be guaranteed at machine-code level too. This makes reasoning about security properties much easier.

Unfortunately an important security requirement has received little attention. Many security properties only hold while the system is up and running. In practice machines crash, go down for reboots and lose power at unexpected moments in time. To account for such events, stateful protected modules must securely store their state. Parno et al. [32] showed that this is a non-trivial task. Sealing a module's state before it is handed over to the untrusted operating system and written to disk, is insufficient. Additional security measures need to be taken to ensure that: (1) a protected module's state can never be rolled back to a previous, stale state, (2) once a module accepted input, it must either (eventually) finish its execution or never advance at all and (3) unexpected loss of power at any moment in time should never result in a system that cannot be resumed after reboot.

State continuity solutions must take the platform specifications and use case at hand into account. Existing solutions [32, 46] rely on an uninterruptible power source (UPS) or risk wearing out non-volatile memory. Such approaches are acceptable in higher-end applications (e.g., in a cloud setting [7, 40]). On other platforms [35, 36] a UPS may not be available on commercial off-the-shelf (COTS) devices and may lead to significant increases in the bill-of-materials (BOM). In such cases wear and tear on non-volatile memory must be minimized to increase longevity of the device. We present a proven-secure solution to state-continuity and show that it can be applied on a large range of platforms.

More specifically, we make following contributions:

- We present Ariadne, a solution to the state-continuity problem that achieves the mathematical lower-bound of only a single bit flip per state update.

- We show that Ariadne can be easily adapted to reduce wear on EEPROM/NAND flash memory.

- We demonstrate that Ariadne can be applied immediately on the SGX/ME platform, without any hardware modifications. This is particularly important as SGX modules (called enclaves) are destroyed when the system is suspended or hibernated. In addition we compare the use of the Intel Management Engine (ME) to the TPM chip to store freshness information and show that no clear winner exists; much depends on the use case and the available hardware.

The remainder of this paper is structured as follows. In the next section we discuss our attack model and the security properties we need to provide in detail. Section 3 builds upon related work and shows that the state-continuity problem can be easily reduced to that of state-continuous storage. Ariadne's algorithm and its optimizations are discussed in Section 4. We evaluate its security in Section 5. Two possible implementations are discussed and evaluated in Section 6. Finally we discuss related work and conclude in Sections 7 and 8.

2 Problem Definition

2.1 Attack Model

Our goal is to provide state-continuity support to protected-module architectures, without (significantly) increasing their attack surface. As such, we assume the following. First, an attacker is able to compromise the complete (untrusted) software stack. As untrusted operating system services are used to store and retrieve module states, this implies that these states can be replayed.

Second, an attacker is able to halt execution at any moment in time because she has complete control over the system's power supply, or because she can launch other attacks leading to similar results. Especially Intel SGX is vulnerable to such attacks. To protect the system from malicious or badly behaving enclaves, execution control is returned to the kernel whenever an interrupt occurs while an enclave is executing. Regardless of how such attacks are executed, we will refer to them as "power-interruption attacks".

2.2 Security Properties

In order to build a secure and reliable system, it is paramount that we are able to provide three security properties. First, we need to provide rollback prevention; an attacker must not be able to provide a stale state of a module and have it accepted as being fresh. Especially in DRM/ERM contexts such security guarantees are important. Consider as an example a document that should only be printed a limited number of times. Such limitations can be guaranteed easily by first checking and decrementing a monotonic counter before the document is printed.

Second, a module's execution needs to be continuous. Once a module accepted input, it needs to eventually finish its execution based on that input and output all com-

puted results, or it must never advance at all. This property is related to rollback-prevention, but is much stricter. Rollback-prevention may guarantee for example that an X509 certificate authority (CA) does not provide two different certificates with the same serial number. But in practice it is also important that every certificate is accounted for; for every serial number the CA should be able to prove which (if any) unique certificate it signed. Failure to do so may break trust [17] that it did not provide rogue certificates.

Third, we must also be able to guarantee liveness of the system. Unexpected crashes or loss of power at *any* moment in time, must not result in a system that will never be able to recover. Note that this is not an availability guarantee. We only consider interrupts in execution that may occur even if the system is not under attack. We cannot guarantee availability: a kernel-level attacker can easily prevent a system from ever resuming its previous state by erasing the fresh state from disk, enter an endless crash-reboot cycle, or erasing the system's boot image.

3 Background: State-Continuous Storage is Sufficient

Related work [32, 46] already showed that the state-continuity problem can be reduced to that of state-continuous storage. We take the same approach. We first introduce `libariadne`, a library providing state-continuous storage in Section 3.1. Sections 3.2 and 3.3 discuss how this library should be used and introduce a running example.

3.1 `libariadne`'s Interface

We provide programmers with the `libariadne` library that can be linked with a protected module. It provides an interface of three functions.

The `void store_state( Blob *blob, String f_format)` function stores the provided data in `blob` in a file on disk. To enable easy recovery of the fresh state, we require that `f_format` is a format string which includes an integer conversion specifier (i.e., "%i"). The library will internally replace the conversion specifier with the value of a monotonic counter. Note that this is only for practical reasons, an attacker changing the filename may prevent the module from ever being resumed, but it will never result in a rollback attack.

The `Blob *retrieve_state( String f_format )` function will attempt to read a file with a matching filename, verify its integrity and freshness and return its content. When this verification step fails for any reason NULL is returned.

In case an attacker deletes the fresh data from disk – or simply when the hard drive got damaged and needed replacement – the fresh data is permanently lost. The `void purge_state( Blob *init, String format )` function can resolve the situation by allowing the programmer to specify an initial, public state of the module. As this results in a loss of any previously stored sensitive data, this operation does not violate state continuity.

3.2 Security Considerations of the `libariadne` Library

With `libariadne` providing state-continuous storage, protected modules writers can easily guarantee rollback prevention and continuous execution [32,46] by ensuring that modules adhere to two principles:

Requirement 1: Store Input Before Processing *Before* processing any input, protected modules must store their current state, with the received input and called entry point.[1] ∎

Requirement 2: Only Deterministic Protected Modules Any source of non-determinism (e.g., `rdrand` instructions) needs to be considered input, and thus following Requirement 1 stored before being used. ∎

These requirements ensure that when a protected-module is interrupted during execution (e.g., due to a power failure), it will restart the computation based on the *same* input when the module is resumed. Since modules are deterministic, it will either reach an identical state as when power was lost, or its execution is interrupted again before it reaches that state.

3.3 Running Example: A PIN-Protected Secret

Consider a module that protects access to a secret. Only when a user presents a valid PIN, will the secret be returned. To mitigate brute-force attacks, we need to be able to guarantee that a user/attacker can make at most three failed attempts before being locked out indefinitely.

The implementation of the module is presented in listing 1. Whenever the module is loaded in memory, the `on_load` function is called implicitly, and an attempt is made to retrieve the last stored state (line 8). If `libariadne`'s `retrieve_state` function finds a freshly stored file, the module's state is restored and execution of the last called entry point is restarted. Eventually the module will end up in the same state as when the module was interrupted.

If on the other hand no matching file can be found, it is corrupted, or it is stale, `retrieve_state` will return

---

[1] Alternatively we could have opted to state-continuously store the state only right before the module returns output. Unfortunately this is hard in practice as any sources of output need to be considered (e.g., calls to unprotected memory, and timing and page-fault channels [59]).

```
1  #include <libariadne/interface.h>
2
3  static int tries_left;
4  static String pin;
5  static String secret;
6
7  void on_load( void ) {
8      Blob *blob = retrieve_state( "state_%i.pkg" );
9
10     if ( blob != NULL ){
11         // restart computation using state & input in blob
12         ...
13     } else
14         reset();
15 }
16
17 void entry_point reset( void ) {
18     Blob *blob = new Blob( &reset || tries_left || pin ||
                  secret );
19     purge_state( blob, "state_%i.pkg" );
20     pin = "0000";
21     secret = "publicly−known secret";
22     tries_left = 3;
23 }
24
25 String entry_point get_secret( String p ) {
26     Blob *blob = new Blob( &get_secret || p || tries_left ||
                  pin || secret );
27     store_state( blob, "state_%i.pkg" );
28
29     if ( tries_left <= 0 )
30         return "Locked out";
31
32     if ( pin == p ) {
33         tries_left = 3;
34         return secret;
35     } else {
36         −−tries_left;
37         return "Incorrect PIN";
38     }
39 }
40
41 bool entry_point set_pin( String p_old, String p_new ){...}
42 bool entry_point set_secret( String p, String s_new ){...}
```

Listing 1: A running example: A PIN-protected secret. The || operator is used to denote concatenation.

NULL. It is up to the module writers to handle such situations. In this case the module is reset to a known good initial state (line 14), at the cost of losing the protected secret indefinitely. The same reset entry point can be called when the user exhausted her 3 access attempts. Note that the reset function stores the previous tries_left, pin and secret within the created blob. While this is not required given that the function will al-
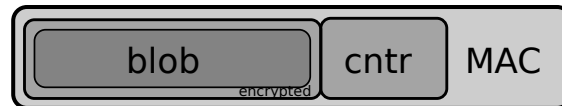

Figure 1: Layout of a package. A module's state and input is confidentiality and integrity protected. The enclosed cntr value enables Ariadne to determine freshness.

ways set these variables to a known-good value, it does *not* enable an avenue of attack; Ariadne will always ensure that the reset function will be called upon recovery.

When on_load successfully retrieved and resumed the fresh state, the user can attempt to access the secret by calling the get_secret entry point and providing it with a PIN. When she hasn't exhausted her number of guesses yet, the provided PIN p is verified (line 32). When the correct PIN is provided, the tries_left variable is reset and the secret is returned. Otherwise tries_left is decremented and an error string is returned. The entry points set_pin and set_secret – offering the obvious functionality to the user – use identical security measures to protect against brute-force attacks.

4 Ariadne

We describe Ariadne in three steps. In Section 4.1 we provide a scheme for state-continuous storage based on a monotonic counter. In Sections 4.2 and 4.3 we propose alternative counter encodings that will lead to a minimal approach to state continuity.

We will only focus on state-continuity guarantees for a single module. Related work [32, 46] showed that support for an unlimited amount of modules can be added easily by (1) using a single state-continuous module to provide secure, state-continuous storage for other modules and (2) inter-module communication. For completeness, we elaborate in Appendix B.

4.1 State-Continuity based on a Monotonic Counter

Guaranteeing state continuity based only on a single monotonic counter to keep track of the fresh state, is a hard problem. Developers of protected modules may be tempted to re-use solutions borrowed from anti-replay security measures, but these are flawed.

Strawman Solutions Generally two approaches are considered. In one approach the monotonic counter is incremented first. Afterwards, the state of the module is confidentiality protected, appended with the counter

value and the whole is integrity protected. Figure 1 displays the resulting *package* graphically. When the system crashes, only the package with an encapsulated counter value equal to the monotonic counter is accepted as being fresh. This approach has the obvious problem that a crash before the new package was written to disk, prevents the system from recovering; liveness cannot be guaranteed.

Alternatively, first storing the package with the next counter value *before* the monotonic counter is incremented also fails. Repeated crashes before the monotonic counter was incremented, enables the creation of multiple packages with the same counter values but different user input. This enables dictionary-style attacks and thus breaks rollback prevention and continuity guarantees. We elaborate on both attacks in Appendix A.

Key Observations Ariadne relies on two important observations. First, the isolation guarantees of protected module architectures ensure that an attacker cannot jump within the middle of a module. During execution we are thus able to assemble guarantees that may not hold when power is lost unexpectedly.

Second, an attacker is extremely restricted in the valid packages she can get access to. The MAC included in the package prevents her from crafting her own packages, or modifying existing ones. To ensure liveness, we need to write new packages to disk before incrementing the monotonic counter. An attacker can abuse this behavior by crashing the system before the counter is incremented. But this implies that at any moment in time, she has at most access to packages with an enclosed counter value smaller than one increment of the monotonic counter.

Ariadne's key insight is that during recovery from a crash, we need to store the fresh package and increment the monotonic counter *twice* before the encapsulated state is resumed.

A Secure Solution Let's re-use the PIN-protected module to describe our solution. Assume that the module is up and running. When a user requests access to the secret by calling `get_secret` (listing 1, line 25), the input and state of the module is placed in a new blob and the `store_state` function is called. Listing 2 displays its implementation. To ensure liveness, a new package is created with the next counter value and stored on disk. Finally the monotonic counter is incremented.

When the module needs to be re-loaded in memory, its `on_load` function is called implicitly (listing 1, line 7) and the `retrieve_state` library function is called to retrieve its fresh state. A package is read from disk and only accepted as being fresh *iff* its MAC value is verified successfully and its enclosed counter value matches with the value of the monotonic counter (listing 2, line 13-19).

```
1 #include <libariadne/interface.h>
2
3 void store_state( Blob *blob, String f_format ){
4   Package *pkg = create_pkg( blob, hwcntr.value() + 1 )
5   hdd.write( pkg, f_format, hwcntr.value() + 1 );
6   hwcntr.inc();
7 }
8
9 Blob *retrieve_state( String f_format ){
10   Package *pkg;
11   Blob *blob;
12
13   pkg = hdd.read( f_format, hwcntr.value() );
14
15   if ( pkg == NULL || !auth( pkg, get_mac_key() ) )
16     return NULL;
17
18   if ( pkg->cntr != hwcntr.value() )
19     return NULL;
20
21   blob = decrypt( pkg, get_enc_key() );
22
23   // ∀pkg ∈ hdd : pkg−> counter ≤ hwcntr.value()+1
24   pkg = create_pkg( blob, hwcntr.value() + 1 );
25   hdd.write( pkg, f_format, hwcntr.value() + 1 );
26   hwcntr.inc();
27
28   // ∀pkg ∈ hdd : pkg−> counter ≤ hwcntr.value()
29   pkg = create_pkg( blob, hwcntr.value() + 1 );
30   hdd.write( pkg, f_format, hwcntr.value() + 1 );
31   hwcntr.inc();
32
33   // ∀pkg ∈ hdd : pkg−> counter ≤ hwcntr.value()
34   // ∀pkg ∈ hdd : pkg−> counter = hwcntr.value() →
        pkg−> contents = blob
35   return blob;
36 }
37
38 void purge_state( Blob * init_blob, String f_format ){
39   hwcntr.inc();
40
41   Package *pkg = create_pkg( blob, hwcntr.value() + 1);
42   hdd.write( pkg, f_format, hwcntr.value() + 1 );
43   hwcntr.inc();
44 }
```

Listing 2: Ariadne: State-Continuity based on a monotonic counter

Now that we determined that the package read from disk is fresh, we can resume the execution of the module. But before doing so, we need to guarantee that no other packages exist with the same counter value. We already observed that an attacker may have packages with an encapsulated counter value of one increment larger than the monotonic counter. Incrementing the monotonic counter twice will thus guarantee that all attacker's pack-

ages are seen as stale in the future. Only incrementing the counter once is not sufficient. We describe an attack against this scheme in Appendix A.3. To guarantee liveness, new packages are written to disk before the monotonic counter is incremented.

To restart a module from a known-good state, `purge_state` can be called (listing 2, line 38). Similar to the `retrieve_state` function, we need to guarantee that when the function returns, there only exists a single package that will be accepted as being fresh. Hence, in this function too we need to increment the monotonic counter twice. However, since this function can always be restarted, liveness is no longer a concern. We can therefore omit storing a new package before we increment the monotonic counter for the first time (line 39).

Skipping Unprocessed Input A careful reader may have noticed that it is possible for an attacker to force the creation of packages with the same enclosed counter value but with different user input. We will show that this does not break state continuity.

Let's use the PIN-protected module again as an example. The attempted attack goes as follows: the attacker calls the `get_secret` function and provides a PIN p. After the module assembled a `Blob` structure containing the current state of the module, the provided input and the entry point used (listing 1, line 27), the `store_state` library function is called. There the attacker crashes the system right before the monotonic counter is incremented (listing 2, line 6). Since a new package was already written to disk, she now possesses a package with enclosed the next value of the monotonic counter and the provided PIN p.

The attacker now has two options. Neither will break state continuity. The first option is to resume the system without any interference. As the newly written package is not seen as being fresh (the monotonic counter in non-volatile memory was not yet incremented) PIN guess p will be discarded. As this guess was never compared to the real PIN code, the attacker did not learn any new information and state-continuity remains guaranteed.

Alternatively, the attacker lets the system reboot but crashes the module immediately after the non-volatile counter is incremented for the first time in the `retrieve_state` function (Listing 2, line 26). When the system now recovers again, the package with PIN guess p will be seen as being fresh. As no input was processed by the module after this guess was made, state-continuity is also guaranteed in this case. As input can only be skipped until the module completes its `retrieve_state` function, repeated crashes during `retrieve_state` will also not break state continuity.

4.2   State-Continuity by Flipping Bits

Related work [32, 46] relied on the irreversibility of hash values to keep track of freshness information. In order to avoid that unexpected loss of power while the hash value is being updated may lead to corrupted non-volatile memory, both approaches required a 2-phase commit protocol. We take a different approach. We use the state-continuity approach based on monotonic counters in the previous section, but use a counter encoding that only requires a single bit flip per state update. In the next section we will show that in practical implementations we can guarantee that these operations can be implemented atomically, and we thus avoid a need of a 2-phase commit protocol altogether. As every solution to state-continuity requires at least a single flip to be recorded, we reached a theoretical lower limit.

Balanced Gray codes provides such an encoding, with the additional benefit that every bit is (almost) equally used. Construction of these codes with arbitrary lengths is non-trivial. In 2008 a constructive proof of their existence was presented by Mary Flahive [14] but to the best of our knowledge never implemented. We present a fast algorithm with fixed, limited, memory consumption.

4.2.1   Terminology

Gray codes ensure that two adjacent code words differ in only a single digit. For example:

$$\{0\underline{0}, \underline{0}1, 1\underline{1}, \underline{1}0\} \tag{1}$$

and

$$\{000, 0\underline{0}1, 01\underline{1}, \underline{0}10, 11\underline{0}, 1\underline{1}1, 101, \underline{1}00\} \tag{2}$$

are two- and three-digit Gray codes, respectively. Moreover, these encodings are cyclic as the same property applies to the last and first code word. The underlined digits in encoding 1 and 2 are the *transition digits* and show which digit is changed in the next code word. The transition count of a digit is the number of times that digit is used. For example, digit 0 in encoding 2 is used twice.[2] The collection of these transition counts is called the transition spectrum of the Gray code (i.e., $(2,2)$ and $(2,2,4)$ for encoding 1 and 2 respectively). When all transition counts are equal (t.i. $2^n/n$ with n the length of the Gray code), the encoding is said to be *uniform* or *completely balanced*. Obviously this can only be achieved when $2^n/n$ is an integer. In other cases *cyclic balanced Gray Codes* can be constructed where the difference of any pair of transition counts is at most two:

$$\forall 0 \leq i, j < n : |TC_n(j) - TC_n(i)| \leq 2 \tag{3}$$

---

[2] We follow the convention that Gray codes start with digit 0 on the left hand side.

4.2.2 Construction

Let's use the construction of a 5-bit balanced Gray code as a running example. The algorithm is displayed in Figure 2. Balanced Gray Codes of length n are constructed recursively from $n-2$ bit balanced Gray Codes. The balanced 2 and 3-bit Gray codes of encoding 1 and 2 are used as base cases. Each iteration consists of three steps. In the first step, a $2^{n-2} \times 2^2$ grid is constructed. The rows are annotated with all $n-2$ digit Gray codes. The number of columns is fixed for any n and columns are annotated with 2-digit codes. A given vertex now represents an n-bit Gray code by concatenating the Gray code of the row and the column. By construction, two (toroidally) adjacent vertices will now represent adjacent Gray codes. Every Hamiltonian cycle found in this grid now represents an encoding, but care needs to be taken to ensure that it is balanced.

In the second step we partition the grid such that the transition counts within each partition can be easily calculated. Partitions are represented as black boxes in Figure 2. Calculation where a new partition needs to be started is discussed in Section 4.2.3. For now note that we will ensure that (1) new partitions will always start at the first and last row and (2) the number of partitions is even.

Finally, the Hamiltonian cycle is constructed starting with Gray code 00000. With the exception of the last two partitions, each partition is traversed in the same way: visit every vertex in the column before moving to the next column. Whenever an edge of a partition is reached, the horizontal/vertical direction is inverted.

Let's take the fourth partition – the first partition containing more than one row – as an example. The partition is started at vertex with Gray code word 01011 when we were in the third column and going from right to left. The graph is continued by first traversing all rows down the current partition. The digit on the edge is the transition digit used to create the next Gray code. When the partition's end is reached (vertex 11011), we move to the next column (vertex 11001) and the rows in the partition are revisited in reversed order. When the start of the partition is reached again (vertex 01001), the last column is selected and each row is again traversed. Finally, the end of the partition is reached (vertex 11000).

The last and second to last partition are constructed differently as shown in Figure 2. We will show that the created notch in the last two partitions ensures that each partition has exactly the same properties.

4.2.3 Computing Partition Sizes

Based on the construction of the Hamiltonian cycle, we can easily derive the transition counts $TC_n(i)$ of each transition digit i of the resulting n-digit Gray code. We
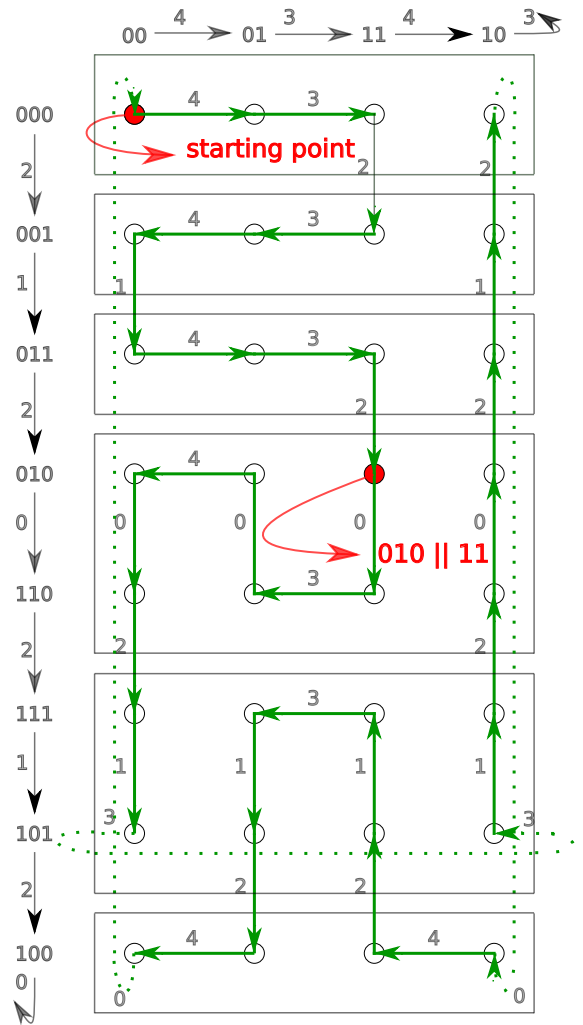

Figure 2: Balanced n-bit Gray Codes can be constructed from $n-2$ balanced Gray Codes

apply a separate reasoning for digits i smaller than $n-2$ (vertical arrows) and digits $n-2$ and $n-1$ (horizontal arrows).

Within a partition vertical arrows are always used 4 times. When a vertical arrow is used as a *connecting digit* – connecting two partitions – it is used twice. This totals the transition counts for these digits to: $4(TC_{n-2}(i) - m_{n-2}(i)) + 2m_{n-2}(i)$ where $m_{n-2}(i)$ is the multiplicity of digit i; the number of times transition digit i is used as a connecting digit.

Transition digits $n-2$ and $n-1$ are represented by horizontal arrows in Figure 2. For standard partitions each is used once per partition. The second to last partition uses $n-2$ twice, but $n-1$ isn't used. In the last partition the opposite happens; $n-1$ is used twice, but $n-2$ isn't used. For the entire graph, transition digits $n-2$ and $n-1$ are thus used L times, with L the number

of partitions. As each partition is connected to the next with a connecting digit, we know that $L = \sum_{i=0}^{n-2} m_{n-2}(i)$. The transition counts for the generated Gray code is thus:

$$TC_n(i) = \begin{cases} 4TC_{n-2}(i) - 2m_{n-2}(i) & \text{if } i < n-2 \\ \sum_{i=0}^{n-2} m_{n-2}(i) & \text{if } i \geq n-2 \end{cases} \quad (4)$$

Flahive [14] showed that a set of connecting digits can always be found such that a balanced Gray code is constructed.

4.2.4 Generating Balanced Gray Codes

Flahive's construction guarantees that we will generate a balanced Gray code when transition digit i is used as a connecting digit between partitions exactly $m_{n-2}(i)$ times. Unfortunately implementing it directly is not possible for large n; keeping track of each partition would quickly consume too much memory. Instead we only precompute $m_{n-2}(i)$ for each $0 \leq i < n-2$ and apply a greedy algorithm that will exhaust each m_i as fast as possible.

Let's reuse the construction of a 5-digit balanced Gray code as an example. Solving equation 4 so that a balanced Gray code is constructed (see equation 3), we get $TC_n(i) = (6,6,8,6,6)$ with $m_{n-2}(i) = (1,1,4)$.

In order to find a graph that fulfills these requirements, we keep track of the size of the current partition, at which row and column we are currently at, in which horizontal and vertical direction we are going and for each transition digit how many connecting digits are still available. Starting at vertex 00000 going in a downwards and right direction (see Figure 2), we calculate (recursively) the transition digit going downward (t.i. 2). As transition digit 2 still needs to be used (4 times) as a connecting digit, we decide to stop the current partition immediately, revert the vertical direction taken and switch columns instead.

Similarly, at vertex 01011 we determine that 0 would be used as a transition digit if we do not terminate the partition. While this transition digit was not yet used as a connecting digit and it should be used once, we do not stop the current partition. Indeed, by construction transition digit 0 is always used as a connecting digit between the last and first partition.

It is important to note that the metadata (in the order of KB) used to generate the next Gray codes, can be included in the package written do disk. Close examination of `libariadne`'s implementation shows that in order to determine whether a package is fresh, we only need to compare the package's counter with the hardware monotonic counter (listing 2, line 18). In other words, we only need access to the Gray codes' metadata after we have already determined that the package is fresh.

4.3 Optimizing for Program-Erase Cycles

Being able to provide state-continuity guarantees by only flipping a single bit per state update, isn't just a theoretical result. It enables various additional optimizations. Let's assume that we use EEPROM/NAND flash memory to store the monotonic counter as an example.

EEPROM is 1980s technology that is still used in some TPM chips [4, 43, 44]. For most applications however it has been replaced by flash memory (e.g., Intel's Management Engine (ME) [37]) because it is cheaper to manufacture for bigger memory sizes. This however comes at a cost; while EEPROM is byte accessible, NAND flash memory needs to be addressed in bigger units. Read and write operations are page-based of usually 2KB to 8KB. Erase commands on the hand operate on blocks of 32-128 pages. [55]

Both EEPROM and flash memory have the disadvantage that they age. Every time the memory is written to or erased, high voltages are applied that eventually will damage the device oxide. Eventually memory cells will be in a stuck-at state, or fail to retain their information over longer periods of time. The number of program/erase (P/E) cycles that can reliably be issued, depends heavily on the manufacturing process (density, single/multi-layer cells, etc.), but typically ranges between 5,000 and 500,000 cycles. [42]

Being provided with different commands to write and erase memory gives us an opportunity to optimize our encoding scheme further. Erasing a block will set all memory cells to 1, while subsequent write commands will set the selected cells to 0. This implies that we can keep issuing write commands until all bits are zeroed out. This significantly reduces the number of erase cycles required and thus increases the memory's longevity.

Our encoding scheme works in two steps and is displayed in Figure 3. First, we apply a Gray code encoding of the monotonic counter used in Section 4.1. In the second step each bit of the resulting Gray code is stored over b blocks of p pages each containing c memory cells and encoded as the number of bits set modulo 2. Each bit flip of the Gray code thus only requires a single write command to one of the pages, which will only touch a single memory cell.

At this low level we must also take unexpected loss of power into consideration; a write/erase command may be interrupted. For write commands this is a non-issue. Since they only affect a single memory cell (t.i. 1 bit), loss of power during their execution will always have a similar effect as a loss of power before or after the command was issued. Both cases are handled at the higher level of the algorithm.

Erase commands in contrast, may not be atomic. Loss of power may leave a memory block in an inconsistent
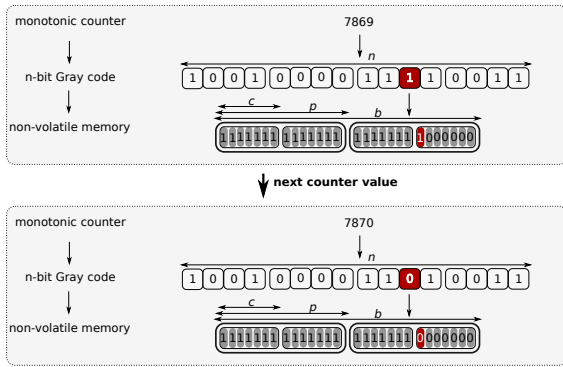
Figure 3: We optimize for write-limited non-volatile memory in two steps: (1) encoding the monotonic counter as an n-bit Gray code and (2) storing each bit in b blocks of p pages each containing c memory cells.

state and gaps in the bit pattern shown in Figure 3 may emerge. As memory content is aggregated to a single bit of a Gray code, such events are at the level of the Ariadne algorithm similar to a crash of the system before or after the counter was incremented. We only need to ensure that each (non-interrupted) write command flips a bit. This can be achieved easily by keeping a local copy of non-volatile memory that is read during the recovery phase.

5 Security Evaluation

Rollback prevention To verify that Ariadne ensures state-continuous execution, we modeled the execution of a module φ:

$$\varphi(State\varphi, Input) \rightarrow State\varphi$$

and proved that even under attack steps, φ is never called with a stale state, or with the same state with different input.

More specifically, we modeled the state of a machine $\mathscr{S}$ as a tuple (H, C, t, P, φ, g) containing a set of packages written to disk H and a monotonic counter C. A term t represents a small program that calls the Ariadne protected module φ in an infinite loop. A set of packages P is kept representing all packages generated by Ariadne. Ghost state g keeps track of (g_{state}, g_i), the last module state and input that was provided to φ as input.

Based on the state space $\mathscr{S}$, we built a state machine with a step relation $S_M \subseteq \mathscr{S} \times \mathscr{S}$ where every step is either a program step or an attack step:

$$S_M = \{(s, s') \in \mathscr{S} \times \mathscr{S} | \text{program\_step s s'}$$
$$\vee \text{inc\_counter\_step s s'}$$
$$\vee \text{modify\_hdd\_step s s'}$$
$$\vee \text{crash s s}\}$$

Program steps simply take one evaluation step of term t. Attack steps on the other hand include:

- `inc_counter_step s s'`: An attacker may advance the monotonic counter. This may prevent the module from ever resuming its state, but it must not break state continuity.

- `modify_hdd_step s s'`: An attacker can modify the contents of the hard disk drive. When the program t reads a package from disk after a crash, it may not receive the last, fresh package.

- `crash s s'`: The system may crash at any point in time. This will immediately reset the program t to its initial state.

Our proof uses rely-guarantee reasoning [20] to show that starting from a known good state s_0 and taking any number of steps in S_M, we will only take allowed steps. A step is allowed when the module φ is called with as input state $(\varphi(s))(g(s))$, the resulting state of the last call to φ. By definition not calling the module $(g'(s) = g(s))$ or purging the module $(g(s') = (s_0, i_0))$ are also allowed:

$$A = \{(s, s') \in S_M | \text{state}(g(s')) = (\varphi(s))(g(s))$$
$$\vee g'(s) = g(s)$$
$$\vee g(s') = (s_0, i_0)\}$$

In total the proof[3] consists of 74 definitions, 74 lemmas and totals 4,823 lines. The optimizations proposed in Sections 4.2 and 4.3 were not modeled.

Liveness property Recall that we also required that an unexpected crash should never let the system end up in a state where it could never advance from. This property is trivially met: the system can only get stuck when it requires access to a package it did not yet store. Since Ariadne ensures that the package with the next counter value is always committed to disk before the counter in non-volatile memory is updated, such situations can never occur.

---

[3]The proof is available for download at `https://distrinet.cs.kuleuven.be/software/sce/ariadne.html`

6 Applications

We already showed that by relying only on a monotonic counter, we can easily optimize our solution, but which optimizations are best applied, depends heavily on the platform and the security guarantees required. We provide two examples, both on the x86 platform.

6.1 TPM NVRAM to Store Freshness Information

On the x86 platform the TPM chip provides an obvious location to store freshness information: it offers many security primitives, is already widely available on commodity devices and is secure against all but sophisticated hardware attacks. Especially the latter is important in a digital rights management (DRM) setting where the client may not be fully trusted, or when the device may get physically stolen by an attacker.

6.1.1 Platform Considerations

Using the TPM, we have two options. We could use the monotonic counters that are directly provided by the TPM to implement the basic Ariadne algorithm from Section 4.1. Unfortunately, in order to ensure that "[the counter does] not wear out in the first 7 years of operation", TPM chips may throttle the speed at which it may be incremented. To comply to the TPMv1.2 specification [53], TPMs only "must be able to increment at least once every 5 seconds." Fortunately TPMs already report on their throttling mechanism, but timeouts between increments may render it unacceptable for some use cases.

Alternatively, we could use TPM NVRAM to store the monotonic counter. While the specification does not require writing operations to be throttled, many TPM implementations rely on EEPROM for NVRAM storage [4,43,44], which may cause repeated NVRAM write operations to "prematurely wear out the TPM." [53] The counter encodings presented in Sections 4.2 and 4.3 can optimize for longevity of this type of memory, but TPM firmware may need to be updated to report the type of memory used, their access granularity and the number of program/erase cycles that are supported.

Independent of whether we use the TPM's native monotonic counters or NVRAM, the number of supported counters is extremely limited. We take the same approach as related work [32, 46] and introduce an indirection. Only a unique *Theseus* module will access the TPM chip directly to store freshness information. Other modules link to a `libariadne_n` library – a slightly modified version of `libariadne` – that uses the Theseus instance to (state-continuously) store a monotonic counter. We elaborate on this construct in Appendix B.

Care must be taken that only a single instance of the Theseus module exists at any point in time. Failure to do so may enable race conditions on the monotonic counter and the same module state may be recovered by multiple module instances. From that point on, an attacker is able to break state continuity trivially by providing different input to the various instances with identical state. Many protected-module architectures [8, 30, 48] allow modules to access their module ID; a unique ID per boot cycle starting at value 0. This makes it trivial to ensure that only a single (Theseus) module exists; if Theseus at initialization-time determines that it received an ID different from zero, it could simply abort.

Intel SGX is a particular case that does not provide such functionality. In this case a static TPM PCR register could be used instead. When the Theseus module starts, it could first extend a static PCR (e.g., PCR 8) with a random value. As static PCRs can never be set to a specific value and only be reset by rebooting the platform, any deviation of the expected resulting value would indicate that a previous instance may have started already.

6.1.2 Implementation

We implemented[4] our prototype on top of Fides [48], an open-source, hypervisor-based protected-module architecture. Table 1 displays the breakdown of the Theseus module and the `libariadne_n` library. It is shown that Ariadne's algorithm is fairly small with only 503 LoCs [56]. The use of balanced Gray codes adds 908 LoCs in total of which 583 LoCs are used to provide static, precomputed metadata. As could be expected, the implementation to balance writes to non-volatile memory is with only 163 LoCs much smaller. Most source code is required to access the TPM chip (1,934 LoCs), perform cryptographic computations (3,237 LoCs), or use basic functions on top of Fides (3,442 LoCs). This results in a total line count for Theseus of 10,399 LoCs. `libariadne_n` is with 7,291 LoCs a bit smaller.

6.1.3 Performance Evaluation

To benchmark TPM operations, `libariadne_n` and the Theseus module, we used a Dell Optiplex 7010 desktop system. It is equipped with an Intel Core i7-3770 CPU (Ivy Bridge) running at 3.40GHz, a TPMv1.2 chip and an SSD drive. It used the generic 3.16.0-31 Linux kernel.

TPM Microbenchmarks We performed benchmarks of various TPM operations (see Figure 4). To force the TPM into a defensive mode and protect itself against possibly wearing out non-volatile memory, we executed

---

[4]Our prototype is available for download at `https://distrinet.cs.kuleuven.be/software/sce/ariadne.html`

libariadne_n	C	x86-64
Fides_tools	1,687	1,755
libcrypto	1,661	1,576
libariadne_base	503	0
other	109	0
Total	*3,960*	*3,331*

Theseus	C	x86-64
Fides_tools	1,687	1,755
libcrypto	1,661	1,576
libariadne_base	503	0
libtpm	1,934	10
libnv_optimize	163	0
libgray_codes	908	0
other	202	0
Total	*7,058*	*3,341*

Table 1: Breakdown of the source code of `libariadne_n` and `Theseus`.

Figure 4: Microbenchmark results (in ms) of various TPM, ME and SSD operations.

every operation 1,050 times. The first 50 timings were later discarded to avoid recording timing results before any defense mechanism kicked in. As expected, commands that do not require access to non-volatile memory, performed well. Polling the TPM for hardware specific information using a `TPM_GetCapability` command finished with only 12.00ms (stdev 2.43) per command significantly faster than any other command. Reading and extending a PCR value took with 24.00ms twice as long (stdev 4.55 and 0.01, resp.).

Other commands such as incrementing a monotonic counter and reading and writing to TPM NVRAM require the creation of an OIAP session. For each benchmark we created a single authorization session and kept the session open for the entire benchmark. Creating and closing a session are thus not included in the measurements, but cost 24.05ms (stdev 0.66) and 23.95ms (stdev 0.01) respectively. Incrementing the same monotonic counter 1,050 times without any interruption between increments, cost with 95.99ms (stdev 5.79) significantly more. To determine whether its performance was throttled to protect against wearing out, we re-executed the same benchmark but with a 5 seconds interval between increments. As expected given that the TPM reports that it does not throttle its speed, the new benchmark provided similar results (95.91ms/inc, stdev 5.41).

We performed similar benchmarks for writing to TPM NVRAM. Each write command only wrote a single zero byte and finished in 144.00ms (stdev 4.26). Introducing a 5 second interval between two write operations did not increase performance and finished in 143.91ms (stdev 3.96). Even though none of these recorded write operations actually required physical writes to TPM NVRAM
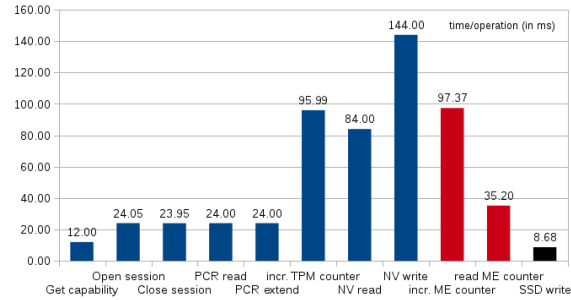
as the same data was provided in every command, reading from TPM NVRAM performed much better at "only" 84.00ms (stdev 2.11). The extremely slow operation of our TPM chips is best illustrated by the performance of the SSD drive. Creating a 1 byte file is with only 8.68 ms (stdev 3.08) significantly faster than accessing TPM NVRAM.

Incrementing a monotonic counter is much faster than writing to TPM NVRAM. We attribute this difference to a wear leveling mechanism within the TPM chip. We tested the presence of such a mechanism on an unused Broadcom TPM chip. We allocated 4 regions of 64 bytes and wrote intermittent `0x00` and `0xff` bytes until these regions failed to retain their contents. Regions broke respectively after 1,450K, 621K, 493K and 301K writes. Without the presence of a wear leveling mechanism, we would have expected that all regions would have failed after an equal amount of write cycles.

In Sections 4.2 and 4.3 we showed that we can minimize the number of program/erase cycles required to implement a counter. Most importantly, we showed that a 2-phase commit protocol to address sudden loss of power during increments is not required. Using these optimizations, we expect that we can further reduce the time required to increment a monotonic counter. We recommend TPM vendors to enable TPM owners to take responsibility of TPM NVRAM wear leveling and expose write/erase commands explicitly.

Benchmarking `Theseus` **and** `libariadne_n` We implemented a state-continuous module that keeps track of a virtual counter. The module was advanced 1,050 times, and we again discarded the first 50 timing results.

When this module accessed TPM NVRAM directly, advancing the counter took 152.01ms (stdev 3.73) in total (see Table 2). As expected, updating freshness information in TPM NVRAM is with 93.68% of the total time by far the most time-costly operation. Calculating the next Gray code, creating the package to store on disk

(in ms)	TPM	SSD	Comp.
Virt. Cntr. (NVRAM)	142.40	9.36	0.25
Virt. Cntr. (Thesius)	145.68	21.91	0.40

Table 2: Benchmark results of `Theseus` and `libariadne_n`

and the overhead introduced by our protected-module architecture, is with 0.25ms negligible.

When the same module used the Theseus module to store freshness information, performance was impacted marginally. Writing the state update of Theseus to TPM NVRAM is with 145.68ms still responsible for 86.72% of the total time required. Communication with the Theseus module caused the time attributed to computation to increase to 0.40ms, but remains negligible. Now that not one but two packages need to be stored on the SSD drive, time lost due to SSD overhead increased from 9.36ms to 21.91ms.

6.2 Intel ME to Provide State-Continuity to Intel SGX

In contrast to earlier provided specifications [18], it became clear with the publication of the Intel SGX SDK [19] in December 2015, that SGX enclaves *can* easily access monotonic counters. It appears that these counters are stored on the management engine (ME) [37], not on non-volatile memory within the CPU package.

Platform Considerations With Intel ME readily available on recent Intel systems it is an interesting location to keep freshness information. Unfortunately it comes with significant downsides compared to an SGX/TPM approach, at least from an academic point of view. First, Intel ME uses a separate processor on the platform control hub (PCH) running its own kernel and processes (e.g., Intel AMT, EPID, etc.). SGX enclaves can access the ME by calling a platform-specific enclave (PSE) that uploads a Java applet through the generic Dynamic Application Loader (DAL) interface [37]. Unfortunately, this re-introduces a TCB of probably a considerable size.

Second, it is unclear how the ME is protected against physical attacks, one of the key selling points of Intel SGX. Related work showed [57] that the ME firmware is only integrity protected and may be accessed through physical attacks [52]. It is unclear how sensitive data stored in the ME is replay protected and whether additional security measures were added in the last generation of PCHs.

Unfortunately Intel SGX/ME does not provide a mechanism for enclaves to determine whether they are the first/only instance. To protect against forking-attacks,

we must make sure that only a single instance of a state-continuous enclave is running. Similarly to the Fides/TPM platform, we could use a PCR register to detect the presence of other enclaves. On systems that lack a discrete, hardware TPM chip, similar functionality is provided by the ME engine.

This approach has the obvious disadvantage that two enclaves need to store their state on disk for every state update. Benchmarks of the Fides/TPM architecture (Section 6.1.3) show that this results in a non-negligible performance impact. This additional overhead could easily be avoided when a (volatile) in-use bit would be kept with the ME monotonic counters. When the enclave state is being recovered, Ariadne should check this bit: when the counter is not in use, the enclave should set this bit – using an atomic `test_and_set` operation – and continue its recovery. When the bit is already set, another instance of the same enclave may still be running and the newly created instance of the enclave should be destroyed to prevent forking attacks. The Theseus module described in Section B uses the same approach.

Benchmarks In the previous section we determined that on the Fides/TPM architecture at least 99.76% of Ariadne's execution time was spent on TPM and disk accesses. We are thus especially interested in the cost of ME monotonic counter increments. We performed microbenchmarks on a recent Dell Inspiron 13 7359 equipped with a Skylake Core i7-6500U processor running Windows 10. As with TPM benchmarks, we discarded the first 50 calls to account for warm-up time for each test. Calling an enclave that returned a static integer value 1,050 times took 0.013ms (stdev 0.003). When the enclave established a single PSE session and incremented a monotonic counter upon each call, performance decreased to 97.38ms (stdev 21.04) with outliers ranging between 71.34 and 251.66ms. Time required to increment an ME monotonic counter is thus comparable to incrementing a monotonic counter in our TPM chip (95.99ms, see Figure 4). When the ME counter was only read, not incremented, performance increased significantly to 35.21ms (stdev 1.17). This leads us to expect that the slow operation of counter increments may be attributed to a throttling mechanism to avoid wearing out ME non-volatile memory.

7 Related Work

Many security architectures have been proposed in recent years. Some rely on a huge TCB making state continuity a much easier problem to solve. Other architectures have ignored the problem altogether and are susceptible to rollback attacks and/or cannot guarantee that the sys-

tem will always be able to advance after power is lost. State-continuity is a problem that has not gained a lot of research attention. Only Parno et al. [32] and we in earlier work [46] proposed solutions to the problem. Others provided more application-specific approaches.

7.1 Systems with a Large TCB

Many systems require state-continuity guarantees in one way or another. These include early designs of protected-module architectures such as Terra [15], AppCores [41] and Proxos [51], but also more conventional systems. Cloud providers must also ensure that the entire state of a virtual machine [16, 58] cannot be rolled back. Similarly, applications on modern operating systems must be restarted from their most recent state.

State-continuity on such platforms can be easily provided as the kernel and/or hypervisor is trusted to isolate disk accesses. It is assumed that an attacker cannot gain access to previously stored states and thus also cannot roll back the state. Since this assumption relies on the correct implementation of the access control logic, this is hard to guarantee in practice. Interestingly, the same applies to many formally verified systems such as seL4 [21], HyperV [24] and XMHF [54]. While the most privileged layer is verified, the file system's implementation usually is not and these systems are still susceptible to attack. Obviously, such designs can also not defend against hardware-level attacks where an attacker is able to physically access the disk to copy and restore stale states.

7.2 Hardware Modifications

Systems such as XOM [26] try to defend against attackers snooping on memory buses by confidentiality and integrity protecting data before it is stored in main memory. Suh et al. [50] show that without anti-replay protection, stale memory contents could be presented as being fresh. Their Aegis architecture and subsequent platforms [9,29] defend against such attacks by including a freshness tag.

Defense mechanisms against memory replay attacks, do not have to take unexpected loss of power into consideration, nor do they have to consider limitations of non-volatile memory. This enables very different approaches.

7.3 Protected-Module Architectures

Many protected-module architectures have been proposed over the recent years. [9,22,27,28,30,31,38,41,48] Interestingly, many do not address the state-continuity problem. To the best of our knowledge only two papers directly addressed the problem. Parno et al. [32] were the first to highlight the problem. They proposed

two mechanisms to provide state-continuity guarantees. Memoir-Basic uses a cryptographic hash to keep track of the fresh state. Unfortunately storing this freshness tag in TPM NVRAM would wear out its memory too quickly to be of any practical use. The authors acknowledge this limitation and propose a solution. Memoir-Opt uses a similar freshness tag, but it relies on TPM PCR registers to protect the most recent value while the system is up and running. When power is lost unexpectedly, an uninterruptible power source ensures that it is written to TPM NVRAM.

In earlier work we proposed ICE [46, 47], an alternative design that assumes "guarded memory": volatile memory within the CPU package that is written to untrusted, non-volatile memory when power is lost. By avoiding TPM/ME accesses for each state update altogether, we achieve better performance results.

Both Memoir and ICE avoid the significantly constrained TPM NVRAM and rely on some kind of uninterruptible power source to provide state-continuity guarantees. While this leads to better performance, these solutions cannot be readily applied in practice. In contrast Ariadne can be used to protect stateful enclaves against rollback attacks on commodity devices.

7.4 Special-Purpose Applications

Many applications have been proposed that provide special-purpose solutions. We can easily provide similar guarantees, but in a much more general way.

Chun et al. proposed a construct called append-only memory [10] to prevent nodes in a distributed network from making conflicting claims to different nodes. A practical implementation was left as future work. In subsequent work Levin et al. [25] provided similar guarantees by only relying on a trusted incrementer (TrInc) that is able to locally store attestation request of monotonic counters.

Schellekens et al. [39] addressed the problem of limited TPM NVRAM. They show that sensitive data can be stored in non-volatile memory off-chip. A light-weight authentication protocol ensures a secure channel between the trusted module and untrusted, non-volatile memory. However, a monotonic counter needs to be incremented for each write instruction and stored in the modified TPM chip. How this counter can be stored efficiently, is not discussed. We achieve similar results without requiring any modification to the TPM chip.

Kotla et al. [23] propose a system to enable offline use of sensitive data. Once sensitive data is accessed, it cannot be denied by the user. Alternatively, if the user attests that the data was never accessed, she will no longer be able to do so in the future. Interestingly, only a very limited TCB needs to be trusted.

The Intel SGX SDK Manual for Windows [19] also discusses how enclaves can be used to implement limited-use policies. Their approach is similar to the inc-then-store discussed in Section A.1. While their approach guarantees that enclave states cannot be rolled back, they fail to guarantee continuous execution of an enclave and liveness of the system. We provide stronger guarantees in a more general way.

8 Conclusion

Protected module architectures enable protected module writers to guarantee formally provable security properties of their code while the system is up and running. But without support for state-continuity, stateful modules are prone to rollback attacks.

Existing solutions relied on the irreversibility property of hash functions and required an uninterruptible power source or risked wearing out non-volatile TPM NVRAM.

We presented Ariadne, the first solution that achieves state continuity based on a counter. By relying on balanced Gray codes to encode this counter, we achieved the theoretical lower limit of requiring only a single bit flip per state update.

Embedded devices can use Ariadne to minimize their total bill of materials. On SGX-enabled x86 platforms Ariadne can be readily applied by relying on the TPM's monotonic counters, TPM NVRAM or the management engine's monotonic counters. We showed that the choice depends on the specific TPM chip on the platform, attack model and use case.

Acknowledgments

References

[1] AGTEN, P., JACOBS, B., AND PIESSENS, F. Sound modular verification of C code executing in an unverified context. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)* (Jan. 2015).

[2] AGTEN, P., STRACKX, R., JACOBS, B., AND PIESSENS, F. Secure compilation to modern processors. In *Proceedings of the 25th Computer Security Foundations Symposium* (Los Alamitos, CA, USA, 2012), CSF'12, IEEE Computer Society, pp. 171–185.

[3] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), vol. 13 of *HASP'13*, ACM.

[4] ATMEL. At97sc3204. http://www.atmel.com/images/atmel-5295s-tpm-at97sc3204-lpc-interface-datasheet-summary.pdf.

[5] AVONDS, N., STRACKX, R., AGTEN, P., AND PIESSENS, F. Salus: Non-hierarchical memory access rights to enforce the principle of least privilege. In *Security and Privacy in Communication Networks (SecureComm'13)* (Sept. 2013), T. Zia, A. Zomaya, V. Varadharajan, and M. Mao, Eds., vol. 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer International Publishing, pp. 252–269.

[6] AZAB, A., NING, P., AND ZHANG, X. 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* (2011), CCS'11, ACM.

[7] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)* (2014).

[8] BRASSER, F., EL MAHJOUB, B., SADEGHI, A.-R., WACHSMANN, C., AND KOEBERL, P. Tytan: Tiny trust anchor for tiny devices. In *Proceedings of the 52Nd Annual Design Automation Conference* (New York, NY, USA, 2015), DAC'15, ACM.

[9] CHAMPAGNE, D., AND LEE, R. Scalable architectural support for trusted software. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture* (2010), HPCA'10, IEEE Computer Society, pp. 1–12.

[10] CHUN, B.-G., MANIATIS, P., SHENKER, S., AND KUBIATOWICZ, J. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP'07, ACM, pp. 189–204.

[11] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. Cryptology ePrint Archive, Report 2015/564, 2015.

[12] EL DEFRAWY, K., AURÉLIEN FRANCILLON, D., AND TSUDIK, G. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *Proceedings of the Network & Distributed System Security Symposium* (Feb. 2012), NDSS'12.

[13] EVTYUSHKIN, D., ELWELL, J., OZSOY, M., PONOMAREV, D., GHAZALEH, N. A., AND RILEY, R. Iso-X: A flexible architecture for hardware-managed isolated execution. In *47th Annual IEEE/ACM International Symposium on Microarchitecture* (Dec. 2014).

[14] FLAHIVE, M. Balancing cyclic R-ary Gray codes II. *The Electronic Journal of Combinatorics 15* (2008), R128.

[15] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Operating Systems Review* (New York, NY, USA, 2003), vol. 37 of *OSR'03*, ACM, pp. 193–206.

[16] GARFINKEL, T., AND ROSENBLUM, M. When virtual is harder than real: security challenges in virtual machine based computing environments. In *Proceedings of the 10th conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2005), HOTOS'05, USENIX Association, pp. 20–25.

[17] HOOGSTRATEN, H., PRINS, R., NIGGEBRUGGE, D., HEPPENER, D., GROENEWEGEN, F., WETTINCK, J., STROOY, K., ARENDS, P., POLS, P., KOUPRIE, R., MOORREES, S., VAN PELT, X., AND HU, Y. Z. Black Tulip - report of the investigation into the DigiNotar certificate authority breach. Tech. rep., FoxIT, 2012.

[18] INTEL CORPORATION. *Software Guard Extensions Programming Reference*, 2013.

[19] INTEL CORPORATION. *Intel Software Guard Extensions Evaluation SDK for Windows OS*, 2015.

[20] JONES, C. Tentative steps toward a development method for interfering programs. In *ACM Transactions on Programming Languages and Systems (TOPLAS)* (New York, NY, USA, 1983), vol. 5, ACM, pp. 596–619.

[21] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), SOSP'09, ACM.

[22] KOEBERL, P., SCHULZ, S., SADEGHI, A.-R., AND VARADHARAJAN, V. Trustlite: a security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys'14, ACM, p. 10.

[23] KOTLA, R., RODEHEFFER, T., ROY, I., STUEDI, P., AND WESTER, B. Pasture: secure offline data access using commodity trusted hardware. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (2012), OSDI'12, USENIX Association, pp. 321–334.

[24] LEINENBACH, D., AND SANTEN, T. Verifying the microsoft hyper-v hypervisor with vcc. In *FM 2009: Formal Methods*. Springer, 2009, pp. 806–809.

[25] LEVIN, D., DOUCEUR, J. R., LORCH, J. R., AND MOSCIBRODA, T. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation* (Berkeley, CA, USA, 2009), vol. 9 of *NSDI'09*, USENIX Association, pp. 1–14.

[26] LIE, D., CHANDRAMOHAN, T., MARK, M., PATRICK, L., DAN, B., JOHN, M., AND MARK, H. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (2000), vol. 35 of *ASPLOS'00*, ACM, pp. 168–177.

[27] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy* (Washington, DC, USA, May 2010), S&P'10, IEEE Computer Society, pp. 143–158.

[28] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems* (Apr. 2008), EuroSys'08, ACM.

[29] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), HASP'13, ACM, p. 8.

[30] NOORMAN, J., AGTEN, P., DANIELS, W., STRACKX, R., HERREWEGE, A. V., HUYGENS, C., PRENEEL, B., VERBAUWHEDE, I., AND PIESSENS, F. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium* (2013).

[31] OWUSU, E., GUAJARDO, J., MCCUNE, J., NEWSOME, J., PERRIG, A., AND VASUDEVAN, A. OASIS: on achieving a sanctuary for integrity and secrecy on untrusted platforms. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), CCS'13, ACM, pp. 13–24.

[32] PARNO, B., LORCH, J. R., DOUCEUR, J. R., MICKENS, J., AND MCCUNE, J. M. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2011), S&P'11, IEEE, pp. 379–394.

[33] PATRIGNANI, M., AGTEN, P., STRACKX, R., JACOBS, B., CLARKE, D., AND PIESSENS, F. Secure compilation to protected module architectures. In *Transactions on Programming Languages and Systems (TOPLAS)* (New York, NY, USA, Apr. 2015), vol. 37, ACM, pp. 6:1–6:50.

[34] PATRIGNANI, M., CLARKE, D., AND PIESSENS, F. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13)* (2013), C.-c. Shan, Ed., vol. 8301 of *Lecture Notes in Computer Science*, Springer International Publishing, pp. 176–191.

[35] RAJ, H., SAROIU, S., WOLMAN, A., AIGNER, R., COX, J., ENGLAND, P., FENNER, C., KINSHUMANN, K., LOESER, J., MATTOON, D., NYSTROM, M., ROBINSON, D., SPIGER, R., THOM, S., AND WOOTEN, D. fTPM: A firmware-based TPM 2.0 implementation. Tech. Rep. MSR-TR-2015-84, Microsoft, Nov. 2015.

[36] RAJ, H., SAROIU, S., WOLMAN, A., AIGNER, R., COX, J., ENGLAND, P., FENNER, C., KINSHUMANN, K., LOESER, J., MATTOON, D., NYSTROM, M., ROBINSON, D., SPIGER, R., THOM, S., AND WOOTEN, D. fTPM: A firmware-based TPM 2.0 implementation. In *Proceedings of the 25th USENIX security symposium* (Aug. 2016), SSYM'16, USENIX Association.

[37] RUAN, X. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*, 1 ed., vol. 1. Apress, 2014.

[38] SAHITA, R., WARRIER, U., AND DEWAN, P. Protecting Critical Applications on Mobile Platforms. *Intel Technology Journal 13*, 2 (June 2009), 16–35.

[39] SCHELLEKENS, D., TUYLS, P., AND PRENEEL, B. Embedded trusted computing with authenticated non-volatile memory. In *First International Conference on Trusted Computing and Trust in Information Technologies (TRUST'08)* (2008), P. Lipp, A.-R. Sadeghi, and K.-M. Koch, Eds., Lecture Notes in Computer Science, Springer-Verlag, pp. 60–74.

[40] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy data analytics in the cloud using SGX. In *36th IEEE Symposium on Security and Privacy* (May 2015), IEEE Institute of Electrical and Electronics Engineers.

[41] SINGARAVELU, L., PU, C., HÄRTIG, H., AND HELMUTH, C. Reducing TCB complexity for security-sensitive applications: three case studies. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems* (New York, NY, USA, 2006), EuroSys'06, ACM, pp. 161–174.

[42] SOLID STATE STORAGE INITIATIVE. NAND flash solid state storage for the enterprise – an in-depth look at reliability. `http://www.vikingtechnology.com/uploads/NV_DIMM_ROI.pdf`.

[43] STMICROELECTRONICS. St19np18-tpm. `http://www.bdtic.com/DownLoad/ST/ST19NP18-TPM.pdf`.

[44] STMICROELECTRONICS. St33tpm12lpc. `http://datasheet.octopart.com/ST33ZP24AR28PVSP-STMicroelectronics-datasheet-16348175.pdf`.

[45] STRACKX, R., AGTEN, P., AVONDS, N., AND PIESSENS, F. Salus: Kernel support for secure process compartments. In *Endorsed Transactions on Security and Safety* (2015), vol. 15, ICST.

[46] STRACKX, R., JACOBS, B., AND PIESSENS, F. ICE: A passive, high-speed, state-continuity scheme. In *Annual Computer Security Applications Conference* (2014), ACSAC'14.

[47] STRACKX, R., JACOBS, B., AND PIESSENS, F. ICE: A passive, high-speed, state-continuity scheme (extended version). CW Reports CW672, Department of Computer Science, KU Leuven, September 2014.

[48] STRACKX, R., AND PIESSENS, F. 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* (New York, NY, USA, October 2012), CCS'12, ACM, pp. 2–13.

[49] STRACKX, R., PIESSENS, F., AND PRENEEL, B. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *Security and Privacy in Communication Networks (SecureComm'10)* (2010), S. Jajodia and J. Zhou, Eds., vol. 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer Berlin Heidelberg.

[50] SUH, G. E., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing* (New York, NY, USA, 2003), ICS'03, ACM, pp. 160–171.

[51] TA-MIN, R., LITTY, L., AND LIE, D. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), OSDI'06, USENIX Association, pp. 279–292.

[52] TERESHKIN, A., AND WOJTCZUK, R. Introducing ring -3 rootkits. In *Black Hat USA* (July 2009).

[53] TRUSTED COMPUTING GROUP. *Design Principles Specification Version 1.2*, 2011.

[54] VASUDEVAN, A., CHAKI, S., JIA, L., MCCUNE, J., NEWSOME, J., AND DATTA, A. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), S&P'13, IEEE Computer Society, pp. 430–444.

[55] WANG, Y., KEI YU, W., XU, S., KAN, E., AND SUH, G. Hiding information in flash memory. In *Security and Privacy (SP), 2013 IEEE Symposium on* (May 2013), pp. 271–285.

[56] WHEELER, D. A. SLOCCount. http://www.dwheeler.com/sloccount/.

[57] WOJTCZUK, R., AND TERESHKIN, A. Attacking intel BIOS. In *Black Hat USA* (July 2009).

[58] XIA, Y., LIU, Y., CHEN, H., AND ZANG, B. Defending against vm rollback attack. In *42nd International Conference on Dependable Systems and Networks Workshops (DSN-W)* (June 2012).

[59] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *36th Symposium on Security and Privacy* (May 2015).

A State-Continuity Based on a Monotonic Counter: Strawman Attempts

State continuity is a hard problem. Solutions borrowed directly from anti-replay defenses fail to provide the required security guarantees. We discuss three approaches.

A.1 Inc, then Store

Listing 3 displays a first – but flawed – implementation. Its implementation is straightforward. Recall the PIN-protected module from Section 3.3. When a user tries to retrieve the module's secret, she calls the `get_secret` entry point and supplies a PIN. Before this

```
1 #include <libariadne/interface.h>
2
3 void store_state( Blob *blob, String f_format ){
4   hwcntr.inc();
5   hdd.write( create_pkg(blob, hwcntr.value()), f_format );
6 }
7
8 Blob *retrieve_state( String f_format ){
9   Package *pkg = hdd.read( f_format, hwcntr.value() );
10
11  if ( pkg == NULL || !auth( pkg, get_mac_key() ) )
12    return NULL;
13
14  if ( pkg->cntr != hwcntr.value() )
15    return NULL;
16
17  return decrypt( pkg, get_enc_key() );
18 }
19
20 void purge_state( Blob * init_blob, String f_format ){...}
```

Listing 3: Incrementing the counter before writing the package to disk will fail to guarantee liveness.

PIN is checked, the module collects the state of the module and passes it together with the provided PIN and called function, to the `store_state` library function. There the (hardware) monotonic counter is incremented first (listing 3, line 4). Afterwards a new package is created containing the incremented counter value and written to disk.

When the system is rebooted and the module needs to be reloaded in memory, the `on_load` function is called implicitly (listing 1, line 7) which in turn calls the `retrieve_state` library function. Next, the library searches for the filename of the last package written to disk. When this package is read and its integrity verified, the counter value enclosed in the package is compared to the hardware monotonic counter (listing 3, line 9 - 14). If both counter values match, the package is determined fresh. After decryption, it is returned to the `on_load` module function where the module's state is restored and the execution of the called function is restarted (listing 1, line 11). If the read package from disk is not fresh or it's integrity check failed, `NULL` is returned and the module is `reset` losing the stored secret indefinitely.

Attack 1: Breaking Liveness
Unfortunately, the provided scheme is flawed. Imagine an unexpected loss of power immediately after the monotonic counter was incremented (i.e., after listing 3, line 4). When the module is reloaded, the `retrieve_state` library function will only accept packages containing a counter value equal to the hardware counter as being fresh. But this package was never stored and the sys-

```
1 #include <libariadne/interface.h>
2
3 void store_state( Blob *blob, String f_format ){
4   hdd.write( create_pkg( blob, hwcntr.value() + 1 ),
        f_format );
5   hwcntr.inc();
6 }
7
8 Blob *retrieve_state( String f_format ){
9   Package *pkg = hdd.read( f_format, hwcntr.value() );
10
11  if ( pkg == NULL || !auth( pkg, get_mac_key() ) )
12    return NULL;
13
14  if ( pkg->cntr != hwcntr.value() )
15    return NULL;
16
17  return decrypt( pkg, get_enc_key() );
18 }
19
20 void purge_state( Blob * init_blob, String f_format ){...}
```

Listing 4: Writing a package to disk before incrementing the monotonic counter will leave the system susceptible to dictionary-style attacks.

tem ends up in a state from where it can never advance. As this situation may also occur even when the system is not under attack, this breaks our liveness requirement. ∎

A.2   Store, then Inc

One may be tempted to quickly fix the design flaw of the scheme presented in the previous section by ensuring that packages are written to disk *before* the hardware monotonic counter is incremented. We will show that also this design does not guarantee the required security properties.

Listing 4 displays the updated scheme. When the module requests the `libariadne` module to store a new state, the `store_state` function will first create a new package with the next counter value and write it to disk (line 4-5) before she increments the hardware counter. The `retrieve_state` function is unchanged.

With these changes in place, a package has always been written to disk that matches the value of the monotonic counter. Therefore the module is always able to recover its state under benign conditions and liveness of this scheme is ensured. Unfortunately, this mitigation enables an attacker to execute a dictionary-style attack.

Attack 2: Dictionary Attack
Let's reuse the module from Section 3.3 as an example. Assume that the attack starts when the module still grants

the attacker with three attempts to guess the correct PIN, and that the monotonic counter has value c. Let's call this state s_0.

In the preliminary step of the attack, the attacker iterates over all entries of her dictionary. For each PIN, she calls the `get_secret` entry point of the module, but crashes the system after the new package – enclosed counter value $c + 1$ – has been written to disk (listing 4, line 4). Since the module could not yet commit to the new input by incrementing the monotonic counter, the module will recover its previous state and once again wait for user input from state s_0. At that point, the attacker continues the same process with the next entry in her dictionary.

When the attacker has reached the end of her dictionary, she finally allows the module to commit to the provided input by incrementing the monotonic value to $c + 1$. Afterwards she crashes the system again.

This marks the start of the second step of her attack. When the module is reloaded in memory, it needs to retrieve its state from disk. Its integrity and freshness is verified before it is used to resume its state. But for every dictionary entry, the attacker possesses a package that will be accepted as being authentic and fresh. She completes her dictionary attack by selecting any package and let the module recover its state based on the enclosed guessed PIN. When she learns the PIN is incorrect, she crashes the system, and selects another "fresh" package. ∎

A.3   Inc once during recovery

In order to defend against dictionary attacks, one must ensure that no other packages with the same fresh counter value exist when the module is being resumed. Incrementing the counter once during recovery does *not* provide this guarantee. We show an attack against such an implementation.

For completeness we show in Listing 5 the (vulnerable) implementation of the state-continuity security measure. The implementation is identical to that of listing 4, but lines 23 to 25 were added. The main issue is that an attacker is able to abuse the creation and storage of packages during recovery (lines 23 and 24) to keep stale packages fresh. This enables attacks similar to the dictionary attack of Section A.2. Since the attack is much harder to grasp completely, we show a simpler version: an attacker is able to provide a guess of the PIN without it being (permanently) recorded, breaking rollback prevention.

Attack 3: Oblivious Steps
For simplicity say that the PIN-protected module is in a state s_0 with 3 PIN-guesses left and the monotonic

counter is at value 42. The attack is executed in 4 steps. In step I an authentic package is created. This means that an attacker provides a guess ``guess``, but crashes the module immediately after it wrote the package `pkg(43 || get_secret || ``guess`` || `$s_0$`)` to disk (listing 5, line 5). As the module only increments the monotonic counter after the state is stored successfully, it still remains at value 42.

Next in step II, the module is recovered to its previous state, leading to a new package `pkg( 43 || ... )`[5] and an incremented counter (listing 5 lines 23 to 25). At this moment in time two different packages with enclosed counter value 43 exist.

In step III the implementation of the `retrieve_state` is abused to create a (soon-to-be-fresh) package `pkg( 44 || ... )` as follows: First, the module is crashed. During recovery the package `pkg( 43 || ... )` is provided and accepted as being fresh. As the `retrieve_state` first writes a new package with an incremented counter value (line 24) the required package is created. Immediately after the package is written to disk, the module is crashed before the monotonic counter can be incremented.

In step IV the module is resumed based on the fresh package `pkg(43 || get_secret || ``guess`` || `$s_0$`)` and the monotonic counter is incremented to 44. At this moment the attacker learns the outcome of her guess. If she guessed wrongly, she can choose to crash the system and let the module recover its state from "fresh" package `pkg(44 || ...)`. As the module didn't record her guess, state-continuity is broken. ∎

B State-continuous storage for *n* modules

`libariadne` as described in Section 4 provides state-continuous storage, at the cost of secure, non-volatile memory. Storing freshness information for every possible protected module in limited-sized, secure non-volatile memory (e.g., TPM NVRAM), is practically infeasible. We resolve the situation using an indirection.

A protected-module "Theseus" is introduced to the system and uses – as the only module in the system – the secure non-volatile memory to store its state. It provides virtual, monotonic counters to other modules executing on the system. Its interface is shown in listing 6.

The `new_counter` entry point to the module, creates a new monotonic counter and protects it with the provided key. It returns the index of the monotonic counter. To

---

⁵We use the notation `pkg( 43 || ... )` here for clarity. We should have introduced a previous state s_{-1}, entrypoint f and input i such that $f(s_{-1}, i) = s_0$. Providing such input to the module will have created the package `pkg( 43 || f || i || s_{-1} )`. As f, i and s_{-1} are irrelevant for the attack, we omit these arguments.

```
1 #include <libariadne/interface.h>
2
3 void store_state( Blob *blob, String f_format ){
4   Package *pkg = create_pkg( blob, hwcntr.value() + 1 )
5   hdd.write( pkg, f_format, hwcntr.value() + 1 );
6   hwcntr.inc();
7 }
8
9 Blob *retrieve_state( String f_format ){
10  Package *pkg;
11  Blob *blob;
12
13  pkg = hdd.read( f_format, hwcntr.value() );
14
15  if ( pkg == NULL || !auth( pkg, get_mac_key() ) )
16    return NULL;
17
18  if ( pkg->cntr != hwcntr.value() )
19    return NULL;
20
21  blob = decrypt( pkg, get_enc_key() );
22
23  pkg = create_pkg( blob, hwcntr.value() + 1 );
24  hdd.write( pkg, f_format, hwcntr.value() + 1 );
25  hwcntr.inc();
26
27  return blob;
28 }
29
30 void purge_state( Blob * init_blob, String f_format ){...}
```

Listing 5: When the counter is only incremented once during recovery, an attacker can force the module to keep stale states fresh, enabling rollback attacks.

```
1 int new_counter( uint64_t key );
2 int counter_set_in_use( int idx, uint64_t key, bool in_use );
3 int counter_increment( int idx, uint64_t key );
4 uint64_t counter_value( int idx, uint64_t key );
```

Listing 6: Theseus' public interface.

protect against inappropriate use, this (index, key)-pair will need to be provided for any subsequent operation on the counter. We assume that communication between the caller and the Theseus module, is confidentiality, integrity and anti-replay protected.

An important feature is the ability to mark a counter to be "in use." This volatile flag, is used to ensure that only a single instance of a protected module can be resumed after a crash. Hence, `counter_increment` and `counter_value` will return an error code if they are called on a counter that was not previously marked as "in use".