Ariadne: A Minimal Approach to State Continuity

Raoul Strackx Frank Piessens

iMinds-DistriNet, KU Leuven, Belgium

August 12, 2016

Outline

1. Introduction
2. Problem Statement
3. `libariadne`
   - Step I: State-continuity based on a monotonic counter
   - Step II: Reducing NVRAM wear & performance overhead
4. Evaluation
5. Conclusion

Protected-Module Architectures

A new approach

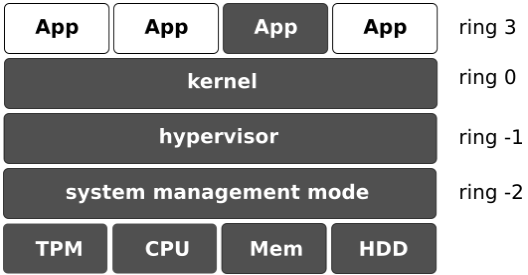
- After ASLR, stack guard, . . . exploits are still rampant
- The TCB is simply too large
- **Main idea:** Protect security sensitive application parts



Protected-Module Architectures

A new approach

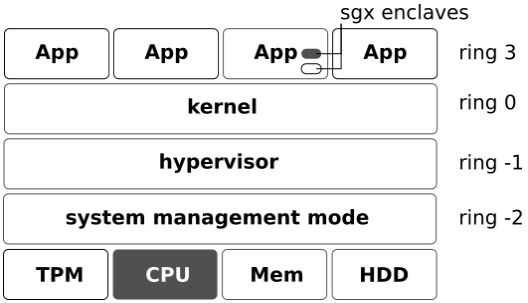
- After ASLR, stack guard, . . . exploits are still rampant
- The TCB is simply too large
- **Main idea:** Protect security sensitive application parts



Protected-Module Architectures

A new approach

- After ASLR, stack guard, . . . exploits are still rampant
- The TCB is simply too large
- **Main idea:** Protect security sensitive application parts



Protected-Module Architectures

Key properties:

- **Isolation**: Program-counter-based access control:
  - Access modules only from within
  - Entry modules through entry points
- **Key derivation**: Unique key per module
- Intel SGX introduced in Skylake (Aug 2015)



"Don't Miss a Sec" art installation by Monica Bonvicini (2004)

Protected-Module Architectures

Key properties:

- **Isolation**: Program-counter-based access control:
  - Access modules only from within
  - Entry modules through entry points
- **Key derivation**: Unique key per module
- Intel SGX introduced in Skylake (Aug 2015)



"Don't Miss a Sec" art installation by Monica Bonvicini (2004)

Protected-Module Architectures

Key properties:

- **Isolation**: Program-counter-based access control:
  - Access modules only from within
  - Entry modules through entry points
- **Key derivation**: Unique key per module
- Intel SGX introduced in Skylake (Aug 2015)

```
205 //secret.c
206 #include "secret.h"
207
208 static int tries_left = 3;
209 static int PIN = 1234;
210 static int secret = 666;
211
212 int ENTRYPOINT get_secret(int provided_pin) {
213   if (tries_left > 0) {
214     if (PIN == provided_pin) {
215       tries_left = 3;
216       return secret;
217     } else {
218       tries_left--;
219       return 0;
220     }
221   } else
222     return 0;
223 }
```

Protected-Module Architectures




```
307 //secret.c
308 #include "secret.h"
309
310 static int tries_left = 3;
311 static int PIN = 1234;
312 static int secret = 666;
313
314 int ENTRYPOINT get_secret(int provided_pin) {
315   if (tries_left > 0) {
316     if (PIN == provided_pin) {
317       tries_left = 3;
318       return secret;
319     } else {
320       tries_left--;
321       return 0;
322     }
323   } else
324     return 0;
325 }
```

Protected-Module Architectures

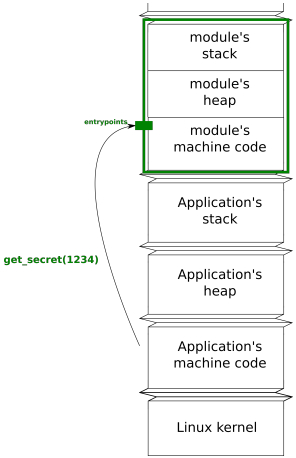
Key properties:

- **Isolation**: Program-counter-based access control:
  - Access modules only from within
  - Entry modules through entry points
- **Key derivation**: Unique key per module
- Intel SGX introduced in Skylake (Aug 2015)

```
428 static int tries_left = 3;
429 static int PIN = 1234;
430 static int secret = 666;
431
432 int ENTRYPOINT get_secret(int provided_pin) {
433   if (tries_left > 0) {
434     if (PIN == provided_pin) {
435       tries_left = 3;
436       return secret;
437     } else {
438       tries_left−−;
439       return 0;
440     }
441   } else
442     return 0;
443 }
```

```
444 void main() {
445   get_secret(1234);
446 }
```

Protected-Module Architectures




```
530 static int tries_left = 3;
531 static int PIN = 1234;
532 static int secret = 666;
533
534 int ENTRYPOINT get_secret(int provided_pin) {
535   if (tries_left > 0) {
536     if (PIN == provided_pin) {
537       tries_left = 3;
538       return secret;
539     } else {
540       tries_left−−;
541       return 0;
542     }
543   } else
544     return 0;
545 }
```

```
546 void main() {
547   get_secret(1234);
548 }
```

Protected-Module Architectures

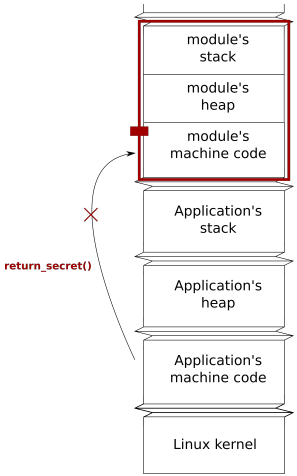
Key properties:

- **Isolation**: Program-counter-based access control:
  - Access modules only from within
  - Entry modules through entry points
- **Key derivation**: Unique key per module
- Intel SGX introduced in Skylake (Aug 2015)

```
651 static int tries_left = 3;
652 static int PIN = 1234;
653 static int secret = 666;
654
655 int ENTRYPOINT get_secret(int provided_pin) {
656   if (tries_left > 0) {
657     if (PIN == provided_pin) {
658       tries_left = 3;
659 return_secret:
660       return secret;
661     } else {
662       tries_left−−;
663       return 0;
664     }
665   } else
666     return 0;}
```

```
667 typedef int (*Func)();
668 void main() {
669   Func get = &return_secret;
670   (*get)(); // Not allowed!
671 }
```

Protected-Module Architectures




```
753 static int tries_left = 3;
754 static int PIN = 1234;
755 static int secret = 666;
756
757 int ENTRYPOINT get_secret(int provided_pin) {
758   if (tries_left > 0) {
759     if (PIN == provided_pin) {
760       tries_left = 3;
761 return_secret:
762       return secret;
763     } else {
764       tries_left−−;
765       return 0;
766     }
767   } else
768     return 0;}
```

```
769 typedef int (*Func)();
770 void main() {
771   Func get = &return_secret;
772   (*get)(); // Not allowed!
773 }
```

Protected-Module Architectures

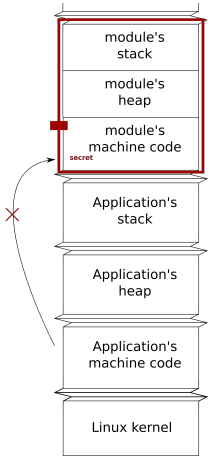
Key properties:

- **Isolation**: Program-counter-based access control:
  - Access modules only from within
  - Entry modules through entry points
- **Key derivation**: Unique key per module
- Intel SGX introduced in Skylake (Aug 2015)

```
876 static int tries_left = 3;
877 static int PIN = 1234;
878 static int secret = 666;
879
880 int ENTRYPOINT get_secret(int provided_pin) {
881   if (tries_left > 0) {
882     if (PIN == provided_pin) {
883       tries_left = 3;
884       return secret;
885     } else {
886       tries_left--;
887       return 0;
888     }
889   } else
890     return 0;
891 }
```

```
892 void main() {
893   printf( "secret = %i\n", secret ); // Not allowed!
894 }
```

Protected-Module Architectures




```
978 static int tries_left = 3;
979 static int PIN = 1234;
980 static int secret = 666;
981
982 int ENTRYPOINT get_secret(int provided_pin) {
983   if (tries_left > 0) {
984     if (PIN == provided_pin) {
985       tries_left = 3;
986       return secret;
987     } else {
988       tries_left−−;
989       return 0;
990     }
991   } else
992     return 0;
993 }
```

```
994 void main() {
995   printf( "secret = %i\n", secret ); // Not allowed!
996 }
```

Protected-Module Architectures

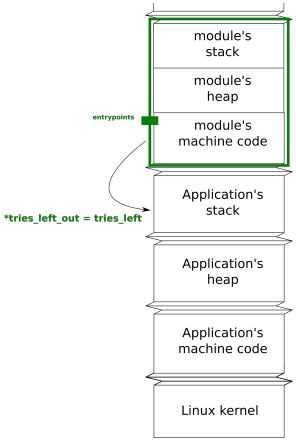
Key properties:

- **Isolation**: Program-counter-based access control:
  - Access modules only from within
  - Entry modules through entry points
- **Key derivation**: Unique key per module
- Intel SGX introduced in Skylake (Aug 2015)

```
1099 static int tries_left = 3;
1100 static int PIN = 1234;
1101 static int secret = 666;
1102
1103 int ENTRYPOINT get_secret(int provided_pin, int *tries_left_out) {
1104   if (tries_left > 0) {
1105     if (PIN == provided_pin) {
1106       tries_left = 3;
1107       *tries_left_out = tries_left;
1108       return secret;
1109     } else {
1110       tries_left--;
1111       *tries_left_out = tries_left;
1112       return 0;
1113     }
1114   } else
1115     *tries_left_out = tries_left;
1116     return 0;
1117 }
```

```
1118 void main() {
1119   int tries_left_out;
1120
1121   get_secret( 1234, &tries_left_out );
1122 }
```

Protected-Module Architectures




```
1201 static int tries_left = 3;
1202 static int PIN = 1234;
1203 static int secret = 666;
1204
1205 int ENTRYPOINT get_secret(int provided_pin, int *tries_left_out) {
1206   if (tries_left > 0) {
1207     if (PIN == provided_pin) {
1208       tries_left = 3;
1209       *tries_left_out = tries_left;
1210       return secret;
1211     } else {
1212       tries_left--;
1213       *tries_left_out = tries_left;
1214       return 0;
1215     }
1216   } else
1217     *tries_left_out = tries_left;
1218     return 0;
1219 }
```

```
1220 void main() {
1221   int tries_left_out;
1222
1223   get_secret( 1234, &tries_left_out );
1224 }
```

Protected-Module Architectures

Key properties:

- **Isolation**: Program-counter-based access control:
  - Access modules only from within
  - Entry modules through entry points
- **Key derivation**: Unique key per module
- Intel SGX introduced in Skylake (Aug 2015)

$$k = kdf(K_{platform}, hash(module_{initial_state}))$$



What we can guarantee:

SW properties of modules (in theory[1]) cannot be broken by SW attacks

[1] Patrignani et al. “Secure compilation to Protected Module Architectures”. 2015. *Transactions on Programming Languages and Systems (TOPLAS)*

What we can guarantee:

SW properties of modules (in theory[1]) cannot be broken by SW attacks

While the system executes continuously

. . . but in real-life systems crash, reboot and power goes down

[1] Patrignani et al. “Secure compilation to Protected Module Architectures”. 2015. *Transactions on Programming Languages and Systems (TOPLAS)*

Outline

1 Introduction

2 Problem Statement

3 libariadne
- Step I: State-continuity based on a monotonic counter
- Step II: Reducing NVRAM wear & performance overhead

4 Evaluation

5 Conclusion

Making the module stateful

```
1327 static int tries_left = 3;
1328 static int PIN = 1234;
1329 static int secret = 666;
1330
1331 int ENTRYPOINT get_secret(int provided_pin) {
1332   if (tries_left > 0) {
1333     if (PIN == provided_pin) {
1334       tries_left = 3;
1335       update_state( new_blob( tries_left || PIN || secret ) );
1336       return secret;
1337     } else {
1338       tries_left−−;
1339       update_state( new_blob( tries_left || PIN || secret ) );
1340       return 0;
1341     }
1342   } else
1343     return 0;
1344 }
```

```
1345 void on_load( void ) {
1346   Blob *blob = recover_state();
1347
1348   if ( blob == NULL )
1349     purge( init_state() );
1350   else
1351     resume_state( blob );
1352 }
1353
1354 void update_state( Blob *blob ) {
1355   EncKey k = get_key( SEAL_KEY );
1356   write_blob( seal( k, blob ) );
1357 }
```

```
1358 void main() {
1359   get_secret(1234);
1360 }
```

Providing State Continuity

```
61 static int tries_left = 3;
62 static int PIN = 1234;
63 static int secret = 666;
64
65 int ENTRYPOINT get_secret(int provided_pin) {
66   if (tries_left > 0) {
67     if (PIN == provided_pin) {
68       tries_left = 3;
69       update_state( new_blob( tries_left || PIN || secret ) );
70       return secret;
71     } else {
72       tries_left--;
73       update_state( new_blob( tries_left || PIN || secret ) );
74       return 0;
75     }
76   } else
77     return 0;
78 }
```

Attack: Providing stale state

- Store the initial state
- Guess the password 3 times
- Crash the system
- Provide the initial state stored on disk as being fresh

state continuity: Requirements

- **No rollback**: prevent the use of stale data
- **Continuous**: output result of provided input, or never advance at all
- **Liveness**: a crash of the machine must not leave the machine unable to advance

state continuity: Requirements

- **No rollback**: prevent the use of stale data
- **Continuous**: output result of provided input, or never advance at all
- **Liveness**: a crash of the machine must not leave the machine unable to advance

How can we reduce strain on access-controled non-volatile memory, without an UPS[2][3]?

[2] Parno et al. “Memoir: Practical State Continuity for Protected Modules”. 2011. *Proceedings of the IEEE Symposium on Security and Privacy*

[3] Strackx, Jacobs, and Piessens. “ICE: A Passive, High-Speed, State-Continuity Scheme”. 2014. *Annual Computer Security Applications Conference*

Outline

1 Introduction

2 Problem Statement

3 libariadne
- Step I: State-continuity based on a monotonic counter
- Step II: Reducing NVRAM wear & performance overhead

4 Evaluation

5 Conclusion

libariadne's interface

```
1 // Throw away all previous blobs, and start from a clean blob
2 void purge( Blob *blob );
3
4 // Update the previously stored blob
5 void update_state( Blob *blob );
6
7 // Recover the previous blob
8 Blob *recover_state( void );
```

A two-step approach

- Step I: State-continuity using a monotonic counter
- Step II: Reducing NVRAM wear & performance overhead

. . . we only discuss state-continuity for a *single* module here

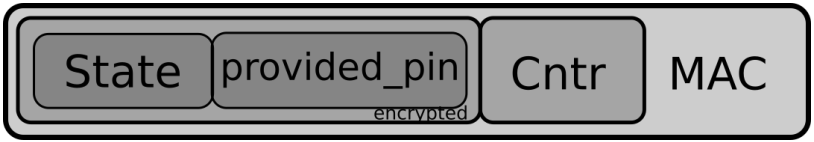
A Library for State-continuous Storage

Reducing the impact of side-channel attacks:

1. Store module's state + input
2. Commit to state update
3. Process input

Requirement: Consider all sources of non-determinism as input

Packages: States stored on disk



Attempt 1

```
1 // Update the previously stored blob
2 void update_state( Blob *b ) {
3   *cntr++
4
5   write_hdd( create_pkg( b, *cntr ) );
6 }
7
8 // Recover the previous blob
9 Blob *recover_state( void )
10 {
11   Package pkg = read_package();
12
13   assert( correct_mac( pkg ) );
14
15   if ( pkg->cntr == *cntr ){
16     return pkg->blob;
17   }
18
19   return NULL;
20 }
```

Attempt 1

```
1 // Update the previously stored blob
2 void update_state( Blob *b ) {
3   *cntr++
4   Crash!
5   write_hdd( create_pkg( b, *cntr ) );
6 }
7
8 // Recover the previous blob
9 Blob *recover_state( void )
10 {
11   Package pkg = read_package();
12
13   assert( correct_mac( pkg ) );
14
15   if ( pkg->cntr == *cntr ){
16     return pkg->blob;
17   }
18
19   return NULL;
20 }
```

- Imagine a crash. . .

Attempt 1

```
1 // Update the previously stored blob
2 void update_state( Blob *b ) {
3   *cntr++
4   Crash!
5   write_hdd( create_pkg( b, *cntr ) );
6 }
7
8 // Recover the previous blob
9 Blob *recover_state( void )
10 {
11   Package pkg = read_package();
12
13   assert( correct_mac( pkg ) );
14
15   if ( pkg->cntr == *cntr ){ No such package exists!
16     return pkg->blob;
17   }
18
19   return NULL;
20 }
```

- Imagine a crash...
- The fresh package was not stored yet!
- Violates liveness requirement

Attempt 2

```
1 // Update the previously stored blob
2 void update_state( Blob *b ) {
3   write_hdd( create_pkg( b, *cntr + 1 ) );
4
5   *cntr++
6
7 }
8
9 // Recover the previous blob
10 Blob *recover_state( void )
11 {
12   Package pkg = read_package();
13
14   assert( correct_mac( pkg ) );
15
16   if ( pkg->cntr == *cntr ){
17     return pkg->blob;
18   }
19
20   return NULL;
21 }
```

Attempt 2

```
1 // Update the previously stored blob
2 void update_state( Blob *b ) {
3   write_hdd( create_pkg( b, *cntr + 1 ) );
4
5   *cntr++
6
7 }
8
9 // Recover the previous blob
10 Blob *recover_state( void )
11 {
12   Package pkg = read_package();
13
14   assert( correct_mac( pkg ) );
15
16   if ( pkg->cntr == *cntr ){
17     return pkg->blob;
18   }
19
20   return NULL;
21 }
```

- The system's state:
  - *cntr = 1000
  - HDD contents:
    - Package((s_0, i_0), 1000)

Attempt 2

```
1 // Update the previously stored blob
2 void update_state( Blob *b ) {
3 →write_hdd( create_pkg( b, *cntr + 1 ) );
4
5   *cntr++
6
7 }
8
9 // Recover the previous blob
10 Blob *recover_state( void )
11 {
12   Package pkg = read_package();
13
14   assert( correct_mac( pkg ) );
15
16   if ( pkg->cntr == *cntr ){
17     return pkg->blob;
18   }
19
20   return NULL;
21 }
```

- Update the state + inc cntr

- The system's state:
  - *cntr = 1000
  - HDD contents:
    - Package((s_0, i_0), 1000)
    - Package((s, i), 1001)

Attempt 2

```
1 // Update the previously stored blob
2 void update_state( Blob *b ) {
3   write_hdd( create_pkg( b, *cntr + 1 ) );
4   Crash!
5   *cntr++
6
7 }
8
9 // Recover the previous blob
10 Blob *recover_state( void )
11 {
12   Package pkg = read_package();
13
14   assert( correct_mac( pkg ) );
15
16   if ( pkg->cntr == *cntr ){
17     return pkg->blob;
18   }
19
20   return NULL;
21 }
```

- Update the state + inc cntr
- Crash

- The system's state:
  - *cntr = 1000
  - HDD contents:
    - Package((s_0, i_0), 1000)
    - Package((s, i), 1001)

Attempt 2

```
1 // Update the previously stored blob
2 void update_state( Blob *b ) {
3 →write_hdd( create_pkg( b, *cntr + 1 ) );
4
5  *cntr++
6
7 }
8
9 // Recover the previous blob
10 Blob *recover_state( void )
11 {
12   Package pkg = read_package();
13
14   assert( correct_mac( pkg ) );
15
16   if ( pkg->cntr == *cntr ){
17     return pkg->blob;
18   }
19
20   return NULL;
21 }
```

- Update the state + inc cntr
- Crash
- Update the state + inc cntr

- The system's state:
  - *cntr = 1000
  - HDD contents:
    - Package((s_0, i_0), 1000)
    - Package((s, i), 1001)
    - Package((s, i'), 1001)

Attempt 2

```
1 // Update the previously stored blob
2 void update_state( Blob *b ) {
3   write_hdd( create_pkg( b, *cntr + 1 ) );
4
5   *cntr++
6   Crash!
7 }
8
9 // Recover the previous blob
10 Blob *recover_state( void )
11 {
12   Package pkg = read_package();
13
14   assert( correct_mac( pkg ) );
15
16   if ( pkg->cntr == *cntr ){
17     return pkg->blob;
18   }
19
20   return NULL;
21 }
```

- Update the state + inc cntr
- Crash
- Update the state + inc cntr
- Crash

- The system's state:
  - *cntr = 1001
  - HDD contents:
    - Package((s_0, i_0), 1000)
    - Package((s, i), 1001)
    - Package((s, i'), 1001)

Attempt 2

```
1 // Update the previously stored blob
2 void update_state( Blob *b ) {
3   write_hdd( create_pkg( b, *cntr + 1 ) );
4
5   *cntr++
6
7 }
8
9 // Recover the previous blob
10 Blob *recover_state( void )
11 {
12   Package pkg = read_package();
13
14   assert( correct_mac( pkg ) );
15
16   if ( pkg->cntr == *cntr ){ True, for both packages!
17     return pkg->blob;
18   }
19
20   return NULL;
21 }
```

- Update the state + inc cntr
- Crash
- Update the state + inc cntr
- Crash
- Upon recovery we can start a “dictionary” attack
- The system’s state:
  - *cntr = 1001
  - HDD contents:
    - Package((s_0, i_0), 1000)
    - Package((s, i), 1001)
    - Package((s, i’), 1001)

Key observations

- We're "collecting" state during execution
- $\forall pkg \in hdd : cntr(pkg) \leq value(cntr) + 1$

Ariadne

```
// Update the previously stored blob
void update_state( Blob *b ) {
  write_hdd( create_pkg( b, *cntr + 1 ) );
  *cntr++
}

// Throw away all previous states, and
// start from a clean state
void purge( Blob *b ) {
  *cntr++
   write_hdd( create_pkg( b, *cntr ) );
  *cntr++
}
```

```
// Recover the previous blob
Blob *recover_state( void )
{
  Package pkg = read_package();

  assert( correct_mac( pkg ) );

  if ( pkg->cntr == *cntr ){
    Blob *blob = pkg->blob;


      write_hdd( create_pkg( blob, *cntr + 1 ) );
      *cntr++


      write_hdd( create_pkg( blob, *cntr + 1 ) );
      *cntr++




      return blob;
  }

  return NULL;
}
```

Ariadne

```
1 // Update the previously stored blob
2 void update_state( Blob *b ) {
3   write_hdd( create_pkg( b, *cntr + 1 ) );
4   *cntr++
5 }
6
7 // Throw away all previous states, and
8 // start from a clean state
9 void purge( Blob *b ) {
10  *cntr++
11   write_hdd( create_pkg( b, *cntr ) );
12  *cntr++
13 }
```

```
1  // Recover the previous blob
2  Blob *recover_state( void )
3  {
4    Package pkg = read_package();
5
6    assert( correct_mac( pkg ) );
7
8    if ( pkg->cntr == *cntr ){
9      Blob *blob = pkg->blob;
10
11     ∀pkg ∈ hdd : cntr(pkg) ≤ value(cntr) + 1
12     write_hdd( create_pkg( blob, *cntr + 1 ) );
13     *cntr++
14
15     ∀pkg ∈ hdd : cntr(pkg) ≤ value(cntr)
16     write_hdd( create_pkg( blob, *cntr + 1 ) );
17     *cntr++
18
19     ∀pkg ∈ hdd : cntr(pkg) ≤ value(cntr)
20     ∀pkg ∈ hdd : cntr(pkg) = value(cntr) →
21                  blob(pkg) = blob
22     return blob;
23   }
24
25   return NULL;
26 }
```

Step II: Storing the Counter

The naive way:

- “normal” binary encoding
- System may crash during a write
- Requires a 2-phase commit protocol

→ Suboptimal use of NVRAM (too much wear)

→ 2x performance overhead to access NVRAM

000
001
010
011
100
101
010
011
111

Step II: Storing the Counter

001
011
010
110
111
101
100
000

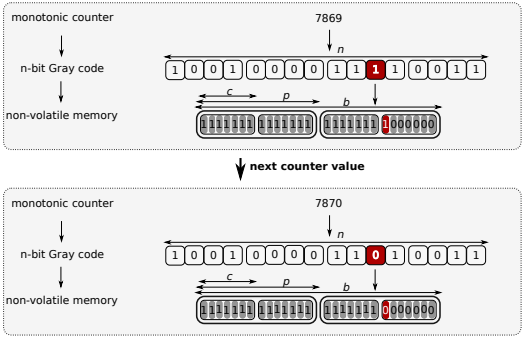
Balanced Gray Codes:

- Only bit one bit per incr.
- Use each bit equally
- Store one bit/storage unit:

$$bit_i = \#bits_set\%2$$

- Crash: previous/next content
- Only one NVRAM write per state update

Step II: Storing the Counter



Balanced Gray Codes:

- Only bit one bit per incr.
- Use each bit equally
- Store one bit/storage unit:

$$bit_i = \#bits_set \% 2$$

- Crash: previous/next content
- Only one NVRAM write per state update

Outline

1 Introduction

2 Problem Statement

3 `libariadne`
- Step I: State-continuity based on a monotonic counter
- Step II: Reducing NVRAM wear & performance overhead

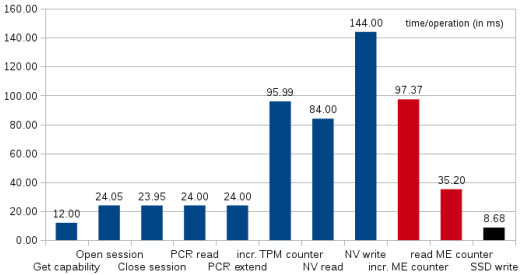
4 Evaluation

5 Conclusion

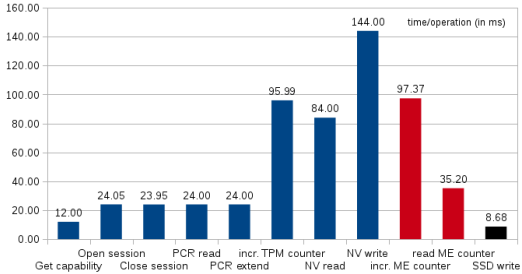
Applications

Intel SGX/TPM

- TPM monotonic counters
  - Speed may be throttled
  - Min inc: one/5 sec
- TPM NVRAM:
  - Access controlled
  - Wears out quickly!
  - Very small
  - Not enough info



Applications



Intel SGX/ME

- Adds ME to the TCB
- No info on wear
- Still quite slow

Outline

1. Introduction
2. Problem Statement
3. `libariadne`
   - Step I: State-continuity based on a monotonic counter
   - Step II: Reducing NVRAM wear & performance overhead
4. Evaluation
5. Conclusion

Conclusion

Ariadne provides state-continuous execution that:

- Is proven secure
- Only requires 1 NVRAM write operation / state update
- Applicable on Intel SGX (Skylake) platforms today
- Applicable for low-end PMAs as well

Questions?