# Memoir: Practical State Continuity for Protected Modules

Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens
*Microsoft Research*
{*parno,lorch,johndo,mickens*}*@microsoft.com*

Jonathan M. McCune
*Carnegie Mellon University*
*jonmccune@cmu.edu*

***Abstract*—To protect computation, a security architecture must safeguard not only the software that performs it but also the state on which the software operates. This requires more than just preserving state confidentiality and integrity, since, e.g., software may err if its state is rolled back to a correct but stale version. For this reason, we present Memoir, the first system that fully ensures the *continuity* of a protected software module's state. In other words, it ensures that a module's state remains persistently and completely inviolate. A key contribution of Memoir is a technique to ensure rollback resistance without making the system vulnerable to system crashes. It does this by using a deterministic module, storing a concise summary of the module's request history in protected NVRAM, and allowing only safe request replays after crashes. Since frequent NVRAM writes are impractical on modern hardware, we present a novel way to leverage limited trusted hardware to minimize such writes.**

**To ensure the correctness of our design, we develop formal, machine-verified proofs of safety. To demonstrate Memoir's practicality, we have built it and conducted evaluations demonstrating that it achieves reasonable performance on real hardware. Furthermore, by building three useful Memoir-protected modules that rely critically on state continuity, we demonstrate Memoir's versatility.**

## I. Introduction

Many security architectures [8, 12, 13, 17, 22–24, 31, 33, 35] attempt to improve system security by isolating the security-sensitive portions of applications and system services. The resulting *protected modules* are pieces of code that operate on opaque state and expose a limited API to the outside world. The modules are protected by a layer of privileged code, e.g., a hypervisor or VMM, that isolates the protected module from all other code (Figure 1), reducing the module's Trusted Computing Base (TCB).

If the protected module and the TCB are correctly written, then *while the TCB continuously executes*, this isolation-based architecture suffices to protect the module's execution. Unfortunately, reboots, machine crashes, or power failures inevitably interrupt the TCB's execution, and indeed, in some systems, the TCB is ephemeral by design [23, 24, 31]. Thus, isolation-based architectures require a secure mechanism for ensuring *state continuity* for protected modules across TCB interruptions. In other words, when the module resumes execution after a TCB interruption, it should be in the same state it was in before the interruption.

Prior work has attempted to provide state continuity in two ways. Some systems [8, 12, 13, 17, 22, 35] accept a bloated TCB that includes code for file systems and storage device drivers. This increases the risk of bugs [26] and provides incomplete protection (see §III-E). Another approach [23, 24, 31, 33] preserves TCB minimality by relying on the untrusted code for persistent storage but encrypting and integrity-protecting a snapshot of the module's state between module invocations. However, prior solutions in this category are either vulnerable [23, 31, 33] to a *rollback attack*, or unable [24, 31, 33] to achieve *crash resilience*.

In a *rollback attack*, untrusted code violates the safety of state continuity by providing the TCB with an old state snapshot. The snapshot is internally consistent and cryptographically sound, but running the module on that snapshot incorrectly ignores the module's execution history. For example, a password verification module might be rolled back to undo a user's password change, or a differentially-private [11] database might be rolled back to violate the database's privacy guarantees. Note that a rollback attack is distinct from a classic replay attack in which the adversary replays a previously seen message to a stateful recipient. In a rollback attack, the recipient's state itself is reverted to a prior state.

Even in the absence of attacks, a lack of *crash resilience* can undermine the liveness of the protected module. In other words, an unexpected crash may leave the protected module in an inconsistent state or prevent it from making forward progress. For example, suppose that a security framework provides a trusted monotonic counter to thwart rollback attacks. When a protected module transitions to a new state, it increments the trusted counter and includes the new counter value in the state that it persists to disk. Upon its next invocation, the module checks whether the counter in its ostensibly fresh state is equivalent to the current value of the trusted counter. This simple technique allows the module to detect valid yet stale states. However, if the computer crashes between the increment of the trusted counter and the persistent write of the new module state, the module becomes unusable. The trusted counter's value will always be at least one greater than the value in any state that the untrusted code can provide. Thus, the module will always refuse to advance further.

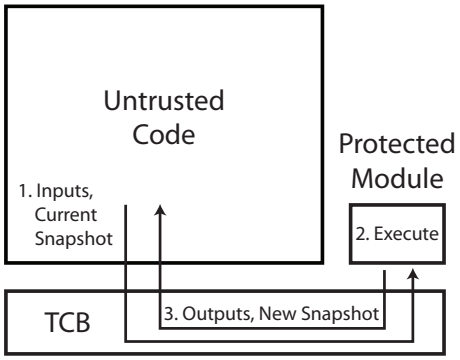


Figure 1. **Execution Model.** *The Trusted Computing Base (TCB) isolates the protected module from untrusted code. Memoir securely maintains state continuity for the protected module, despite relying on the untrusted code for bulk persistent storage.*

In this paper, we describe Memoir, a generic framework for guaranteeing the safety and liveness of state continuity for protected modules. A key insight is that we can achieve these properties by making our protected modules behave deterministically; even non-deterministic modules can be made deterministic via a PRNG and a cryptographically protected seed. In particular, a deterministic module that is rolled back to a previous snapshot will leak no new information *as long as* untrusted code is not permitted to provide the protected module with an input that is *different* from the one it provided the last time the module started from that snapshot. Memoir enforces input consistency by using trusted hardware to store a concise check on the full history of inputs to the module. As a result, legitimate code can always recover from crashes, since it can ensure that the most recent request and snapshot are made persistent before invoking the module. However, malicious code cannot use this recovery mechanism to violate state continuity.

In practice, instantiating Memoir is made challenging by the limitations of commodity secure hardware. In particular, the most widely available commodity secure hardware, the Trusted Platform Module (TPM) [37], has serious performance and resource limitations. For example, although the TPM specification dictates the inclusion of monotonic counters, it only requires the ability to increment a counter once every five seconds, which is clearly insufficient for high-event applications like networked games [20]. Similarly, although the TPM specification mandates access-controlled nonvolatile RAM, most implementations provide only 1,280 bytes of NVRAM. Even worse, the NVRAM is quite slow and is only expected to support 100K write cycles across its entire lifetime; writing once every second would exhaust NVRAM in less than 28 hours.

Memoir circumvents these limitations by avoiding the monotonic counters, minimizing the number of writes to slow, write cycle-limited NVRAM, and using a novel multiplexing technique to support multiple protected modules with a constant amount of NVRAM. With these techniques, Memoir supports an arbitrary number of protected modules while performing NVRAM writes only twice per boot, at which rate we conservatively estimate existing TPMs' NVRAM lasting for over 136 years.

To assess Memoir's security, we develop formal, machine-checked proofs of safety using TLA+ [19]. The proofs include 243 definitions, 74 named theorems, and 5816 discrete proof steps. At a high level, they show that Memoir's protocols preserve state continuity for protected modules.

We implement Memoir on Linux, employing the Flicker architecture [24] to isolate the protected modules. We also create a variety of Memoir applications, ranging from a simple count-limited password checker to a sophisticated differentially-private statistics module. Our evaluation shows that Memoir's state continuity protection adds only 76 ms of overhead to request execution, and that we can save 17 ms of this overhead with our optimization to limit NVRAM writes.

In summary, we make four key contributions.

1) We demonstrate the importance of state continuity for protected modules, highlighting the dangers of rollback attacks and crash-induced inconsistencies.
2) We design a generic, minimal-TCB framework to securely provide state continuity for protected modules.
3) We provide a concrete implementation of this framework that provides high performance despite the limitations of current secure hardware.
4) Using TLA+, we formally prove the correctness and security of our framework.

## II. Problem Definition

### A. Execution Model

In this work, we focus on an execution model (see Figure 1) in which a module consisting of code and security-sensitive data is protected from other code on the system. We refer to the code that is responsible for isolating the protected module as the Trusted Computing Base (TCB). While the TCB may take the form of a hypervisor, virtual machine monitor (VMM), or even the hardware itself, this basic model encompasses a plethora of systems [8, 12, 13, 17, 22–24, 31, 33, 35]. This model is popular, in part, because it allows security-sensitive code operations to coexist safely with potentially buggy legacy code.

We assume that, either by design [23, 24, 31] or as a result of machine reboots or crashes, the TCB is not continuously in control of the platform. As a result, we require a secure mechanism to preserve *state continuity* for the protected module. We can decompose this into a safety property and a liveness property [2]. For safety, if we think of the protected module as a state machine, then we require that adversarial actions should not be able to force the state machine into an invalid state, nor into a valid state via an invalid transition. For liveness, we require that machine crashes should not leave the state machine unable to advance.

To keep the TCB minimal, we primarily focus on architectures [23, 24, 31, 33] that do not include file system and storage device driver code in the TCB. Thus, the protected module must rely on untrusted code for the bulk of its storage needs. We discuss extension of our design to systems with larger TCBs in §III-E.

Finally, we assume some small amount of trusted persistent storage is available to the TCB. We aim to minimize the amount needed, both to reduce hardware costs and to minimize TCB complexity.

### B. Adversary Model

The adversary's goal is to violate the continuity of the protected module's state. Potential violations include placing the module in an invalid state (e.g., convincing a monotonic counter module to use a negative counter value), or causing the module to make an invalid transition to a legitimate state (e.g., convincing a monotonic counter module to roll back to an earlier counter value). However, we do not consider denial-of-service attacks, since the adversary could always choose to turn off the computer.

The adversary can run arbitrary code in the untrusted code environment and has full control over the bulk persistent storage facilities on the machine, meaning he can modify, delete, and retrieve old versions of any data placed in bulk storage. The adversary also controls power to the machine, and hence can cause reboots at arbitrary points in time, even when the TCB controls the platform.

We assume the adversary cannot launch sophisticated hardware attacks (e.g., physically reading memory or examining CPU registers). We also assume the correctness of the TCB and protected module, which implies that the adversary cannot violate the isolation imposed on the untrusted code. Though we briefly discuss some specific side-channel attacks, an exhaustive treatment of all such threats is beyond the scope of the current paper.

### C. Challenges on Current Systems

On current commodity computers, the most widely available secure hardware is the Trusted Platform Module (TPM) [37], a special-purpose security chip which is deployed on over 200 million machines [15]. The TPM includes several Platform Configuration Registers (PCRs), which can record information about the software state of the platform. Via an attestation protocol, the TPM can convey the value of these PCRs to an external verifier; see prior work for details [28, 37].

The TPM also ostensibly includes a number of features that should aid secure state continuity. Unfortunately, in practice, these features have significant limitations. The TPM specification requires TPMs to support at least four monotonic counters, but only requires them to sustain one increment every five seconds, which significantly limits the rate at which a protected module can update its state. Sarmenta et al. look at other limitations and suggest some hardware-based improvements [32].

The TPM also includes a limited amount of nonvolatile RAM (NVRAM). Reading and writing to NVRAM can be restricted based on the contents of the PCRs, so an NVRAM location can be made accessible only to a single module. Unfortunately, the specification only requires 1,280 bytes of NVRAM, some of which is dedicated to various system features such as storing the TPM's endorsement credential and the launch control policy. Our experiments (see §VI-B) indicate that although reading NVRAM is relatively fast (9.8–14.8 ms), writing is 3–6x slower (33.9–82.4 ms). In addition, the NVRAM is only expected to tolerate a limited number ($\sim$100,000) of write cycles during its lifetime. Writing to NVRAM once every second would exhaust its write cycles in less than 28 hours.

### D. Strawman Solutions

We now demonstrate the complexity of providing state continuity by illustrating subtle issues with solutions that might otherwise appear to work well.

**Monotonic counter.** A fragile technique for preventing rollback attacks is to include the value of a trusted monotonic counter in the state. As discussed in §I, this approach is not crash resilient, since a crash after the counter is updated but before the new state has been made persistent will render the system unusable. In addition, as discussed above, monotonic counters on current hardware suffer from several limitations.

**Allowing replay.** Crash resilience can be achieved by allowing untrusted code to replay its input after a crash. However, untrusted code can abuse this functionality by claiming (or causing) a crash and then providing the protected module with a different input, hence causing a rollback attack.

**Two-phase commit.** A standard technique for achieving crash resilience is a two-phase commit protocol, in which the coordinator asks each participant to tentatively perform an operation and provide the output. When all participants have provided tentative output, the coordinator tells each participant to commit. However, these protocols are concerned with atomicity of actions, rather than the secrecy and integrity of one party's state. Indeed, in our model, the coordinator is untrusted code and must not see the output from the protected module until the operation is truly committed.

We considered the following variant of a two-phase commit protocol, but it proved to be inefficient and prone to subtle timing attacks. In brief, rather than store a monotonic counter in NVRAM, the protected module stores a hash of the encryption of the latest state and output. The encryption scheme must be randomized but not stateful (e.g., CBC with a random IV). To make a request, the untrusted system supplies the protected module with the request, the current encrypted state, and the last encrypted output. If the hash of the encrypted state and output match the contents of NVRAM, the module performs the request. It uses the TPM

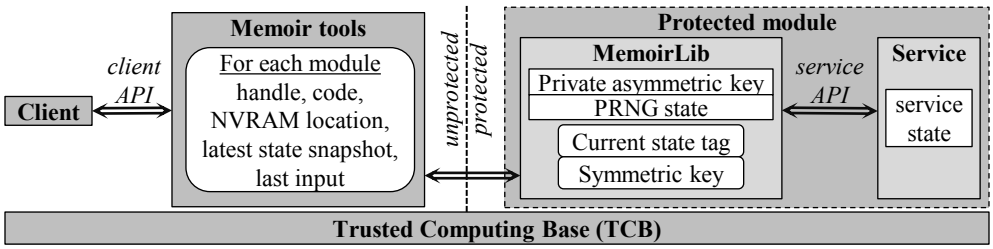


Figure 2. Architecture of Memoir. Rounded boxes indicate persistent storage.

to create a fresh random IV and encrypts the new state and output. It hashes this encryption, but does *not* update its NVRAM. Instead, it simply returns the encrypted state and output to the untrusted system, along with a signature linking the current hash to the tentative next hash.

To commit the request, the untrusted system writes the encrypted state, encrypted output, and signed statement to disk. Then, it passes them to the protected module with an instruction to commit. The protected module verifies the signature is valid, refers to the current hash, and contains a next hash that is the hash of the supplied state and output. If so, it updates its NVRAM to contain the next hash and returns the decrypted output. If a crash happens between the NVRAM update and output revelation, the untrusted system supplies the encrypted state and output to the module and asks for the decrypted output. The module verifies that the encrypted state and output correspond to the hash in NVRAM, to make the revelation safe.

This approach would work, but is inferior in two major ways to Memoir. First, it is expensive: for each request, it requires two invocations of the trusted module, an NVRAM write, and an access to the TPM's slow random number generator. In contrast, Memoir needs only one invocation of the trusted module, can avoid NVRAM writes as discussed in §III-C, and can avoid the TPM's random number generator as discussed in §V-A. Second, and most seriously, two-phase commit allows a subtle rollback attack based on side channels. When the adversary submits a request, he receives the resulting encrypted state and output. Thus, he can observe side channels, such as the size of the resulting state and output, and the time to perform the request [1]. Based on these side channels, he can then decide whether to commit the request; not doing so is essentially equivalent to rolling it back. This attack might be used, for instance, to make unlimited incorrect password guesses without causing the module to enter a state where it backs off requests.

**Computation on encrypted data.** Instead of using a protected module, we might instead employ one of the many cryptographic techniques for allowing an untrusted party to perform an operation over encrypted data [29]. However, such techniques tend to be limited in the functionality they offer and/or exorbitantly expensive from a performance perspective, whereas protected modules allow arbitrary functionality to operate directly on the raw data.

## III. Memoir Design

Memoir is our system for providing state continuity without sacrificing crash resilience. Since it is agnostic to the protection framework used, and since TCB minimality is a key goal, we make only the most basic assumptions about the storage protection afforded by that framework. §III-E discusses optimizations enabled by larger-TCB protection systems that provide greater storage protection.

We propose two main designs for Memoir, which we call Memoir-Basic and Memoir-Opt. The latter is optimized to avoid NVRAM writes and thereby achieve better performance and longevity, but it requires an Uninterruptible Power Supply (UPS). This is because although Memoir-Opt can survive OS crashes, it cannot handle unexpected power failures.

§III-A gives an overview of Memoir's architecture, then §III-B and §III-C describe Memoir-Basic and Memoir-Opt. We then discuss potential extensions of our design, to support any number of coexisting modules (§III-D) and to make use of facilities in larger-TCB protection systems (§III-E). Finally, §III-F suggests a small TPM modification to further enhance Memoir.

### A. Architecture

Figure 2 illustrates the architecture of our system. Untrusted clients use untrusted Memoir tools to interact with protected modules. A protected module is implemented by linking a *service* with MemoirLib, Memoir's module library. A service is a piece of code that handles requests and produces responses, without concern for state continuity; it links with MemoirLib to ensure continuity of its state.

MemoirLib ensures state continuity by mediating all interaction with the untrusted system, in particular leveraging the untrusted system to provide storage. It does so by outputting state *snapshots* and verifying and decrypting them when they are passed back as input. Figure 3 illustrates the contents of a snapshot. It contains an encrypted serialization of all the module's state, including both the service state and MemoirLib state. It also contains a *freshness tag*, used by Memoir to determine whether a snapshot is up-to-date. Finally, it contains an authenticator, which takes the form: $\mathsf{MAC}_{\mathsf{SymmetricKey}}(\text{“AUTH”}||\mathsf{EncryptedState}||\mathsf{FreshnessTag})$. MemoirLib uses this to determine a snapshot's legitimacy.

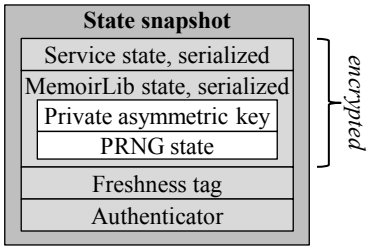


Figure 3. Contents of a state snapshot

Memoir's design requires that the service be *deterministic*. This is reasonable, since the main sources of nondeterminism are time, random numbers, and multithreading [21, 30]. Time can be supplied as an input to a service and, if needed, it can be signed by a trusted time source. To provide randomness, MemoirLib includes a cryptographically-secure deterministic PRNG (pseudo-random number generator) [40]. Memoir does not currently support multithreaded modules, since we imagine most will be single-threaded for the sake of simplicity and small TCB. However, if needed, techniques exist for making multithreaded programs deterministic [4, 5, 27].

To be protected, a service must implement four functions: initialize its state in memory, handle a request to produce a response, serialize its state, and deserialize its state. MemoirLib uses these functions when creating modules and executing requests. We omit discussion of how MemoirLib allows remote verification of a module's public key, since our approach is standard [24] and unrelated to the problem of ensuring state continuity.

Figure 4 shows in detail how Memoir creates a module. At a high level, the client supplies the Memoir `create` tool with the module's code (1). For instance, he might supply code for a password-checking module. The `create` tool is responsible for basic bookkeeping and interfacing with the protected module (2-3, 8-9). Once the module has been instantiated and initialized (4-7), the `create` tool (10) returns a handle, which the client can use when performing additional operations with the module.

Figure 5 shows the details of how Memoir executes a request. (1) The client invokes the Memoir `execute` tool, supplying a module handle and a request to pass to that module. For instance, the request might be to check password `P@55w0rd`. The tool manages the state snapshots and interfaces with the protected module (2-3, 9-10). MemoirLib is responsible for performing various security checks and invoking the service (4-8). Finally (11), the `execute` tool gives the response, for example, `password incorrect`, to the client.

Our approach to defending against adversarial denial-of-service is to require root access for the files and TPM capabilities that the Memoir tools use. Thus, normal users are unable to make modules inaccessible by, e.g., deleting snapshots or deallocating NVRAM regions. However, by giving the tool binaries the setuid attribute, normal users can perform legitimate operations on modules. It might seem odd to place trust in the root user and the OS since the design is meant to take the OS out of the TCB. However, note that the trust is only that they will not deny service to local users, which they could do anyway in various other ways. A root compromise cannot cause a module to operate incorrectly, roll back state, or mislead verifiers; it can only prevent users from invoking a module.

In general, Memoir defines the identity of a protected module to be a hash of its code. However, applications that wish to support upgradeable modules can do so using Memoir's primitives. As with all upgrade systems, to upgrade securely, the module developer must include a mechanism by which the old module can authenticate the new module, e.g., by providing the old module with a public key and signing a hash of the new module.

To begin the upgrade process, the application asks the Memoir tools to instantiate the new module. The old module can then perform a "remote" verification of the new module's identity and public key [24], using the Memoir tools as an untrusted intermediary. Assuming the verification succeeds and the identity matches the certified value for the new module, the old module does two things: (1) It transitions to a terminal state, from which it will refuse to process further requests. (2) It outputs its state encrypted under the new module's public key. The atomicity of this state transition plus output are guaranteed by Memoir's design. Hence, the upgrade process is itself protected from problems of continuity safety and liveness.

### B. Achieving State Continuity: Memoir-Basic

Recall from §II-D that if a module uses a monotonic counter value as its freshness tag, then a crash can render it unusable. The problem is that the system may crash between the time the module updates the protected counter, and the time the untrusted system receives and durably stores the new state snapshot. Once this happens, the user can only supply snapshots to the module that will be rejected as being more stale than the current value of the counter.

Our solution to this is to tag a state not with the *number* of requests that led to it but with the *sequence* of requests that led to it. We call this sequence the state's *history*. This method of tagging is useful because it gives us the ability, after a system crash, to replay the last request without compromising security. Since we can be assured that the request being replayed is identical to the last request performed, we can safely re-execute the request. Because the module is deterministic, it will do the same thing during replay that it did originally. Hence, it will not reveal any more information to the untrusted system than has already been revealed.

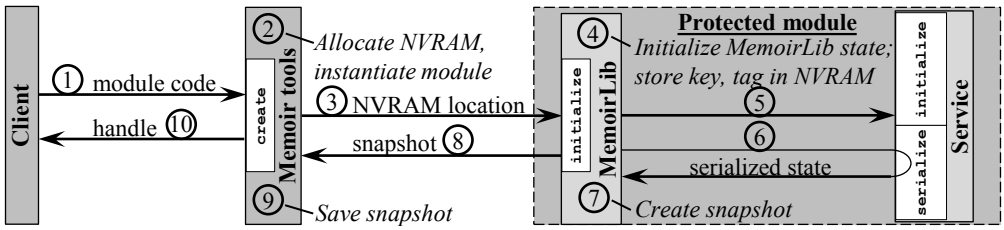


Figure 4. Module creation in Memoir: (1) The client runs the `create` tool on the module's code; this code includes the service and MemoirLib. (2) The tool allocates an NVRAM location and instantiates the module. (3) The tool calls the module's `initialize` routine, passing it the NVRAM location. (4) The routine checks that the NVRAM location's protection is adequate, initializes a PRNG, creates keys, and stores the symmetric key and initial freshness tag in the NVRAM location. (5) It then calls the service's `initialize` routine to initialize the service's state and (6) calls the `serialize` routine to obtain a serialization of this state. (7) It incorporates the service's state into its own, then encrypts and tags the aggregate state. (8) It returns the resulting snapshot to the tool. (9) The tool saves the snapshot to disk, then (10) returns to the client a handle to use when referencing the module.

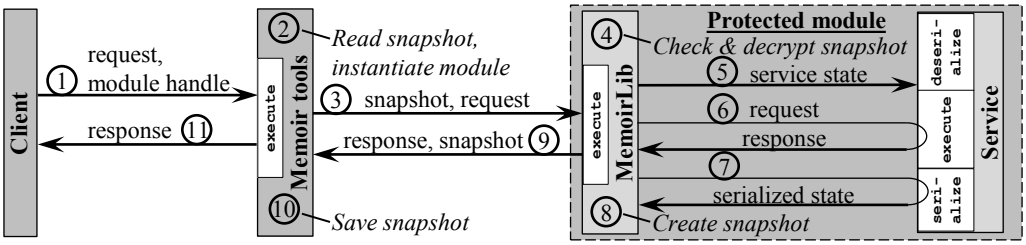


Figure 5. Request execution in Memoir: (1) The client runs the `execute` tool, giving it a request and a handle to the module to execute that request. (2) The tool looks up the module, reads its latest snapshot from disk, and instantiates the module. (3) The tool calls the module's `execute` routine, passing it the snapshot and request. (4) The routine reads the symmetric key and freshness tag from NVRAM and uses them to check that the snapshot is valid and fresh. If so, it decrypts the snapshot. (5) It calls the service's `deserialize` routine, passing it the service's serialized state from the snapshot. (6) It calls the service's `execute` routine to execute the request and obtain a response. (7) It calls the service's `serialize` routine to serialize the resulting state. (8) It incorporates the service's state into its own, then encrypts and tags the aggregate state. (9) It returns the resulting snapshot to the tool. (10) The tool saves the snapshot to disk, then (11) returns the response to the client.

We can securely summarize a module's history with a hash chain. We define the initial history summary by $\mathsf{HistorySummary}_0 = 0$, and let the history summary after the $n$th request $r_n$ be $\mathsf{HistorySummary}_n = \mathsf{Hash}(\mathsf{HistorySummary}_{n-1} \parallel r_n)$. The properties of a secure hash make it hard for an adversary to produce multiple request sequences that produce the same summary. Thus, by using the history summary as the freshness tag and storing that in protected NVRAM, we protect the module against rollback while only using a few bytes of this scarce resource.

Using the history summary as our freshness tag also allows us to safely recover from crashes, as we now illustrate. Suppose that the system crashes after a module performs request $r_n$ but before the untrusted system can persist the resulting snapshot $\mathsf{Snapshot}_n$. The module now expects $\mathsf{Snapshot}_n$ but the untrusted system does not know it; it knows only $\mathsf{Snapshot}_{n-1}$ and $r_n$. The untrusted system would like to supply these to the module and have it replay the request, producing the lost snapshot $\mathsf{Snapshot}_n$ as well as the lost output. The history summary allows the module to safely do so.

From the module's perspective, it has been given an allegedly-previous snapshot $\mathsf{AllegedPrevSnapshot}$ and an allegedly-previous request $\mathsf{AllegedLastRequest}$. The module can check the authentication and freshness tag in the alleged snapshot and determine that it corresponds to a particular history summary $\mathsf{AllegedSummary}$. If $\mathsf{Hash}(\mathsf{AllegedSummary} \parallel \mathsf{AllegedLastRequest})$ matches the current trusted history summary, then the module can be assured the claim is true. It is thus safe to replay this request on the supplied snapshot, thereby recovering from the crash.

All that remains is to ensure that the untrusted system can supply the previous state snapshot and last input in the event of a crash. Fortunately, this is straightforward, because the Memoir tools store them durably on disk before invoking the protected module. In the event of a crash, it has them readily available for use in recovering the system.

It is important that MemoirLib updates the freshness tag in NVRAM *before* invoking the trusted module. This is to prevent leakage of information through the side channel of execution time. To illustrate, suppose an adversary wants to check a password without updating the module's state. Suppose further that the adversary knows the module takes at least time $t$ if and only if the password is incorrect. The adversary can invoke the module, wait for time $t$, then crash the system if he has not heard a response. If MemoirLib had not already updated the freshness tag, the adversary would not have to replay the same request he did before; he could behave as if it never happened.

### C. Avoiding NVRAM Writes: Memoir-Opt

A limitation of Memoir-Basic is that it requires a slow update of write-cycle-limited NVRAM on every request. For this reason, we now present our high-performance variant Memoir-Opt that rarely writes to NVRAM but still maintains full security. However, it requires a checkpoint routine to take place before the system shuts down, necessitating special precautions to deal with crashes. To deal with system crashes, we must program the hardware to invoke the checkpoint routine before shutting down; this is possible because the checkpoint routine is a protected, OS-independent module that requires no access to disk. To deal with power failures, we need to attach an Uninterruptible Power Supply (UPS) to give Memoir-Opt warning that it needs to invoke the checkpoint routine.

Because Memoir-Opt is complex, we describe it in stages. First, we describe the TPM feature it relies on, the PCR. Second, we describe how we can use a *two-part history summary* to store the history summary in both NVRAM and a PCR, to avoid common-case NVRAM writes. Third, we discuss how to use a *guard bit* and *extension secret* to protect against adversaries who can reset the computer without triggering the checkpoint routine. Finally, we discuss the practicalities of setting up a checkpoint routine.

**Using a PCR to avoid NVRAM writes.** To avoid NVRAM writes, Memoir-Opt requires trusted memory that will persist across module invocations. For this, we use a PCR. The property of a PCR we rely on is that it can only be updated in two ways: (1) rebooting to set it to 0, or (2) *extending* it [37]. One extends a PCR by supplying a 20-byte value $h$, causing the PCR's contents to become $\mathsf{Hash}(\mathsf{OldValue} \parallel h)$.

Our standard design assigns one PCR to each module, though §III-D will show how to use one PCR for all modules. To ensure the module knows which PCR to use, we store the PCR index in one byte of NVRAM, by the symmetric key.

**Two-part history summary.** To achieve crash resilience, we cannot rely on a PCR alone, since the PCR is cleared upon shutdown. Instead, we will store the history summary in a combination of NVRAM and the PCR. For this, we use a *two-part* history summary, $\langle \mathsf{NvramPart}, \mathsf{PcrPart} \rangle$. To avoid NVRAM writes, we craft a summary update function that rarely updates NvramPart. To deal with shutdowns, we have our checkpoint routine condense the entire summary into only NvramPart so that PcrPart can be zeroed.

For our initial history summary, we use $\langle 0, 0 \rangle$. To compute the next history summary following $\langle n, p \rangle$ given a request $r$, we use $\mathsf{Successor}(\langle n, p \rangle, r)$, defined as:

$$\langle n, \mathsf{Hash}(p \parallel \mathsf{Hash}(\mathsf{ExtensionSecret} \parallel r)) \rangle.$$

ExtensionSecret is a secret known only to the module; we will discuss its purpose later. The *checkpoint* of a history summary is defined by:

$$\mathsf{Checkpoint}(\langle n, p \rangle) = \begin{cases} \langle n, p \rangle & \text{if } p = 0 \\ \langle \mathsf{Hash}(n \parallel p), 0 \rangle & \text{otherwise.} \end{cases}$$

Note that these definitions are specifically adapted to the TPM's PCR-extension operation discussed above.

(1) To determine if a tag $s_{\mathsf{alleged}}$ is fresh:
- Set $s$ to the current history summary
- If $s = s_{\mathsf{alleged}}$, return true
- If $s = \mathsf{Checkpoint}(s_{\mathsf{alleged}})$, return true
- Otherwise, return false

(2) To determine the current history summary:
- Read NvramPart and GuardBit from NVRAM
- Read PcrPart from the PCR
- If GuardBit $= 1$ and PcrPart $= 0$, return *invalid* (malice detected)
- If GuardBit $= 0$ and PcrPart $\neq 0$, return *invalid* (shutdown required after recent checkpoint)
- Otherwise, return $\langle \mathsf{NvramPart}, \mathsf{PcrPart} \rangle$

(3) To change current history summary from $s$ to $\mathsf{Successor}(s, r)$:
- Extend the PCR with $\mathsf{Hash}(\mathsf{ExtensionSecret} \parallel r)$
- Set GuardBit in NVRAM to 1 if it isn't already

(4) To check if $s_{\mathsf{alleged}}$ and $r_{\mathsf{alleged}}$ are the previous tag and input:
- Set $s$ to the current history summary
- If $s = \mathsf{Checkpoint}(\mathsf{Successor}(s_{\mathsf{alleged}}, r_{\mathsf{alleged}}))$, return true
- If $s = \mathsf{Checkpoint}(\mathsf{Successor}(\mathsf{Checkpoint}(s_{\mathsf{alleged}}), r_{\mathsf{alleged}}))$, return true
- Otherwise, return false

(5) To checkpoint in anticipation of shutdown:
- Set $s$ to the current history summary
- If $s$ is *invalid*, exit
- If GuardBit is already 0, exit
- Set NvramPart in NVRAM to $\mathsf{Hash}(s.n \parallel s.p)$
- Set GuardBit in NVRAM to 0

Figure 6. Pseudocode for Memoir-Opt operations

Figure 6 provides pseudocode showing how we use these two-part history summaries in practice. We now discuss these algorithms in detail, though we defer discussion of the guard bit until later.

When Memoir-Opt is asked to execute request $r$ on a snapshot alleged to be the current one, it does the following. First, it must check whether the snapshot's tag $s_{\mathsf{alleged}}$ is fresh (Figure 6, function #1). To do so, it extracts the current history summary $s = \langle \mathsf{NvramPart}, \mathsf{PcrPart} \rangle$ from NVRAM and the PCR (Figure 6, function #2). If $s = s_{\mathsf{alleged}}$ or $s = \mathsf{Checkpoint}(s_{\mathsf{alleged}})$, then the snapshot is fresh. There are two cases here because it is possible there was a checkpoint between the time the snapshot was generated and now. Even if a checkpoint occurred, the snapshot is still fresh.

Next, before executing the request, Memoir-Opt must update the history summary to reflect it (Figure 6, function #3). It does so by updating it to $\mathsf{HistorySummary}_{new} = \mathsf{Successor}(\mathsf{HistorySummary}_{old}, r)$. This only requires extending the PCR part, and never modifies the NVRAM part,

due to the definition of Successor. Thus, we stay within the limits of the PCR's abilities, and avoid writing NVRAM.

Now, consider the case where the untrusted system asks Memoir-Opt to replay alleged last request $r_{\mathsf{alleged}}$ on a snapshot with tag $s_{\mathsf{alleged}}$. Memoir-Opt must check that the given request and snapshot truly constitute the last request and previous snapshot (Figure 6, function #4). To do so, it extracts the current history summary $s$ as described earlier. Then, it checks whether $s = \mathsf{Checkpoint}(\mathsf{Successor}(s_{\mathsf{alleged}}, r_{\mathsf{alleged}}))$ or $s = \mathsf{Checkpoint}(\mathsf{Successor}(\mathsf{Checkpoint}(s_{\mathsf{alleged}}), r_{\mathsf{alleged}}))$. In either case, it is safe to re-execute the request and supply the response. This time, the reason there are two cases is because there may or may not have been a checkpoint after the snapshot $s_{\mathsf{alleged}}$ was taken and before the last request occurred. On the other hand, it is certain that a checkpoint was taken after the last request occurred, for the following reason. The only reason why the untrusted system would ask for a replay of a request is if the system shut down so soon afterward that the untrusted system did not have a chance to write the output to disk. If the system shut down after a request, a checkpoint must have been taken.

To condense the history summary into NVRAM before shutdown, the checkpoint routine computes $\mathsf{Checkpoint}(\mathsf{HistorySummary})$ and stores the NVRAM part in NVRAM (Figure 6, function #5). The PCR part will then be set to its correct value of zero upon shutdown.

**Preventing rollback attacks.** Unfortunately, the approach as we have so far described it is subject to a rollback attack. The adversary can accomplish this by disconnecting the UPS and thereby powering off the machine without triggering the checkpoint routine. If this happens, then the PCR's value will reset to zero without its information being condensed into NVRAM. This allows the adversary to roll back to the snapshot in use when the computer last turned on.

To prevent this, we need an additional bit of NVRAM, which we call the *guard bit*. The module sets this bit upon extending the PCR, to indicate that the PCR's value is invalid if it is zero. The checkpoint routine clears the bit after condensing the PCR's data into NVRAM, to reflect that the PCR's value is valid if zero. MemoirLib never trusts its history summary if it finds the guard bit and PCR inconsistent.

This prevents the rollback attack we described earlier, but another attack is still possible. The adversary can prevent a checkpoint, reset the computer, then extend the PCR with one or more of the requests that followed the previous system power-up. This will put the PCR in a state consistent with an outdated snapshot, which the adversary could present as fresh. Unfortunately, the guard bit will not help us since the PCR is non-zero and the guard bit only prevents the module from incorrectly treating a zero PCR as correct.

This is why we use ExtensionSecret, generated at random at initialization time and stored along with MemoirLib's other state. By using this secret, even though the adversary can readily observe history summaries in the PCR, he cannot learn the inputs that must be supplied to the TPM to get it to produce those summaries. If he could, he could invert the hash function, a cryptographically hard operation. Since the adversary cannot extend the PCR to produce valid history summaries, he cannot roll back to earlier states.

**Setting up the checkpoint routine.** We now briefly discuss the practicality of an OS-independent mechanism that will be reliably invoked during system shutdown, even if the shutdown is the result of an OS crash or a power failure. The most logical place to install such a routine is as a System Management Interrupt (SMI) handler that executes in a special CPU mode—System Management Mode (SMM) [16]. SMM is an operating mode with full device and memory access that can preempt all code running in Protected Mode, even hypervisors. The principal challenge in installing a Memoir-Opt shutdown handler as an SMI handler is to modify the code that runs in SMM. The most reasonable approach is to collaborate with a BIOS vendor. The HyperSentry project showcases some of the interesting capabilities of custom SMI handlers [3]. With the SMI handler in place, the SMI itself can be triggered via a communication from an uninterruptible power supply to the IPMI server management facility. Another option is to use existing low-level OS mechanisms that can generate SMIs in software, e.g., the Linux kernel's `BUG` facility.

### D. Allowing Unlimited Modules

One concern with our design is that it limits the number of modules that can coexist on a machine. This is because each module requires scarce NVRAM, enough to store a hash and symmetric key. The problem is even worse for Memoir-Opt, which also requires a PCR exclusively devoted to each module. Fortunately, we have an approach that allows any number of modules to coexist while using only as much protected memory as is necessary for one module.

Our approach is to have only one module use protected nonvolatile memory: the *module-management module* ($M^3$). Each other protected module uses a version of MemoirLib that relies on the $M^3$ for freshness protection and symmetric key storage, and thus does not need any NVRAM allocated to it. We call a module that relies on the $M^3$ in this way a *submodule*; as a result, the $M^3$ is in the TCB of each submodule. Figure 7 summarizes the $M^3$'s operation.

Like any module, the $M^3$ handles requests and produces responses. However, these requests are essentially meta-requests, e.g., "create submodule S" and "submit request $r$ to submodule S." Before we discuss how the $M^3$ handles these requests, it will be helpful to describe what state it keeps for each submodule and how it can invoke submodules. Then, we will describe how the $M^3$ handles its meta-level requests. Finally, we will discuss a mechanism to prevent submodules from denying service to other submodules despite their shared reliance on the $M^3$.

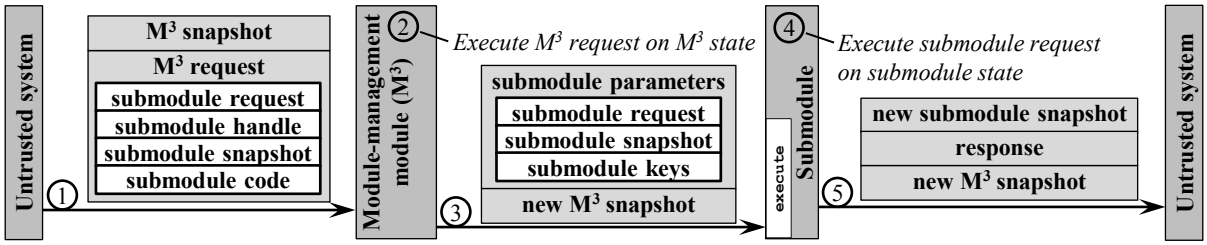


Figure 7. Submodule request execution via the module-management module ($M^3$): (1) The untrusted system executes an $M^3$ request, which involves sending the $M^3$ an $M^3$ snapshot and $M^3$ request. (2) The $M^3$ executes the request as much as it can without invoking the submodule. If the submodule information is valid, it increments the history length for the submodule. (3) It now loads the submodule's code and jumps to its `execute` routine, passing in various parameters, as well as the new $M^3$ snapshot. (4) The submodule executes the submodule request. (5) The submodule returns to the untrusted system its new snapshot and its response to the request. It also returns the new $M^3$ snapshot it received earlier.

| Handle | Code hash | Keys | History length |
|---|---|---|---|
| 1 | E0B3... | 8D98... | 15 |
| 2 | 66FF... | B345... | 3 |
| 3 | 1408... | 84D3... | 104 |

Figure 8. An example of the state $M^3$ maintains for each submodule. "Keys" refers to the submodule's symmetric key and asymmetric key pair.

**The $M^3$ state.** For each submodule, the $M^3$'s state consists of the following: a handle to identify it, a hash of its code, its symmetric key, its asymmetric key pair, and its history length, i.e., the number of requests it has executed. An example $M^3$ state is shown in Figure 8. Note that the $M^3$'s state need not include the submodule's history summary, since all submodule requests are implicitly part of $M^3$'s protected history. Thus, submodules only need to tag snapshots with the current history length, instead of the current history summary.

**Invoking submodules.** The $M^3$ can, given a submodule's code as an input, jump to it and supply inputs to it. Before doing so, it clears its secrets from memory and extends a dynamic measurement PCR with the submodule code's hash. The latter prevents the submodule from impersonating the $M^3$ or accessing its NVRAM location.

**Creating submodules.** To create a submodule S, the client supplies the $M^3$ with S's code as input. The $M^3$ then creates a new handle, symmetric key, and asymmetric key pair for S, and adds the relevant information to its state. It then uses its PRNG to generate a seed. Finally, it jumps to S's code, supplying it an input consisting of the command *initialize*, the created keys, and the seed. S uses the seed to initialize its own PRNG.

**Submitting requests to submodules.** To submit $r$ to submodule S, the client follows the procedure illustrated in Figure 7. That is, it supplies the $M^3$ with S's handle, S's code, a snapshot from S, and $r$. The $M^3$, having access to S's code hash, symmetric key, and history length, can check that the code is valid and that the snapshot is authentic and fresh. If so, the $M^3$ increments S's history length, produces a new $M^3$ snapshot, then jumps to S's code. It supplies it with S's snapshot and keys, as well as the request $r$.

To re-execute the last submodule request, a client simply replays its last "submit request" request to the $M^3$. Since the $M^3$ uses Memoir-Basic or Memoir-Opt to ensure state consistency for its own state, it supports re-execution of requests, including re-execution of an $M^3$ request that executes a submodule request.

**Obtaining a submodule's public key.** To get an attested public key for submodule S, a client first gets an attested public key for $M^3$. It then gets an attestation, signed by the $M^3$'s public key, linking S's public key to S's code hash.

**Preventing denial-of-service by a submodule.** One wrinkle is that since the $M^3$ jumps directly to a submodule, it cannot directly pass output to the untrusted system. It must supply that output as an additional input to the submodule, and rely on the submodule to include that output along with its own output, as illustrated in Figure 7. This means that one faulty submodule can cause denial of service to all other submodules by eliding or corrupting the $M^3$'s snapshot in its own output.

To deal with this issue, we add the following capability. The untrusted system can readily figure out that it is the victim of an uncooperative submodule from the fact that the $M^3$ produces an error output when it is subsequently invoked. In this case, we allow the untrusted system to invoke the $M^3$ in a special way, telling it to repeat the last $M^3$ request *but suppress any submodule output*. This is safe, since it gives the untrusted system less information than it could legitimately ask for. When the $M^3$ is invoked this way, it needs not jump to the submodule to obtain any output. Thus, the $M^3$ can give its output directly to the untrusted system, thereby allowing it to continue using the remaining non-faulty submodules, as well as to delete the faulty submodule.

### *E. Using Larger-TCB Systems*

Up to this point, we have been assuming a minimal-TCB protection system that provides limited functionality to trusted modules. However, our approach can be generalized and made useful even on protection systems that include a larger TCB.

**Using enduring protected volatile memory.** If the TCB includes an operating system or virtual machine monitor, it can be relied on even when the protected module is not running. Thus, it can extend protections on volatile memory beyond the duration of a protected module's invocation. This memory is not much use in assuring crash resilience, since it goes away whenever the system shuts down or resets. Thus, all the novel protections of Memoir are still necessary: deterministic modules, secure and concise history storage, and protection against malicious rollbacks disguised as reboots. However, whether using Memoir-Basic or Memoir-Opt, Memoir can use the enduring memory to improve performance by keeping the module's state in memory across invocations.

In particular, Memoir can skip most of the steps of execution depicted in Figure 5. It can skip steps 2, 4, 5, 7, 8, and 10; and, it does not have to transmit state snapshots in steps 3 and 9. Although these steps do not have to be performed on every request, they still need to be performed occasionally. As we will now discuss, we need these steps to obtain periodic snapshots and to recover from shutdowns and resets.

When the system resumes after a shutdown or reset, Memoir will need to restore the module's state. This is why Memoir's techniques remain relevant: they enable restoration without allowing an adversary to misuse the procedure to perform a rollback attack. To restore state, the Memoir tools present to MemoirLib the latest snapshot and, since there may have been more than one request since then, the *full sequence* of requests that have happened since the snapshot. Now, MemoirLib can check that the snapshot and requests represent the true history by running:

$s \leftarrow \mathsf{AllegedSnapshotTag}$
For each supplied alleged request $r$, in order presented:
  $s \leftarrow \mathsf{Hash}(s, r)$
If $s \neq \mathsf{CurrentHistorySummary}$, reject

If the check succeeds, MemoirLib restores the module's state by deserializing the snapshot and replaying the requests.

Since replay takes a long time if the sequence of requests is long, the Memoir tools should have a policy for obtaining periodic snapshots, e.g., obtain one after every 100th request.

**Using unprotected, nonvolatile storage.** If the TCB includes components such as disk drivers and file systems, a protected module may have access to a large amount of unprotected, nonvolatile storage. If this is the case, then Memoir would not ever have to pass state snapshots back and forth between the unprotected system and module. It could write them to disk from within the module itself. However, it would still have to use all of Memoir's techniques to ensure the freshness of this data, as an adversary could overwrite snapshots on disk with old ones.

**Using software-protected, nonvolatile storage.** If the TCB is larger still, it might protect large, nonvolatile storage even when the protected module is not running. For instance, the TCB might include an OS or VMM, plus a facility like secure boot that ensures no other system can be loaded. However, even in this case, Memoir would still be useful in defending against a *disk-cloning attack*. To prepare for this attack, an adversary physically disconnects the disk and copies some of its contents to another disk. To perform the attack, he rolls back the protected module's storage on disk to its earlier contents. The freshness assurance provided by Memoir would detect this attack and prevent it from working.

### F. Suggested TPM Improvement

The main limitation of Memoir is that it requires access to slow, write-cycle-limited NVRAM on every request. Memoir-Opt fixes this, but at the cost of requiring a UPS for crash resilience. If we could change one thing about the TPM, it would be to make NVRAM fast and write-cycle-unlimited. A practical way to achieve this is via a small write-back cache.

By this, we mean that the TPM should include a small amount of access-controlled RAM that functions as a write-back cache of NVRAM. It would then also need a capacitor, to store enough energy to ensure the RAM gets flushed to NVRAM when the power goes out. By doing this, the TPM would achieve the same non-volatility that NVRAM has today, but with much faster access times, and with write-cycle limitations that constrain only the number of shutdowns rather than the number of writes.

With the multiple-module design of §III-D, Memoir only needs enough such fast-writable NVRAM to store a hash. Thus, only a tiny amount of RAM and capacitance would be needed to effect this change.

Incidentally, the TPM specification already requires at least one volatile register to be flushed upon power-down, namely the volatile state register used in random number generation [37]. Since our suggestion requires a very similar capability, we believe it could be practically implemented.

## IV. Formal Proofs of Safety

To ensure the correctness of our design, we developed formal safety proofs using TLA+ [19]. Although the proofs were constructed manually, they are mathematically precise, and they have been programmatically machine-verified using the TLA+ Proof System [7]. The proofs include 243 definitions, 74 named theorems, and 5816 discrete proof steps. The formal specifications and proofs are published in a 390-page tech report [10] as a companion to the present paper.

We follow a proof approach encouraged by TLA+: Rather than asserting a particular set of safety properties that happen to occur to us, we define a high-level specification that describes the intended semantics of the system. Since Memoir is a platform that supports arbitrary services, our high-level spec declares the service to be an undefined function that maps a state and a request to a state and a response. In

TLA+, state changes are defined by *actions*, and our high-level spec contains one main action, AdvanceState, which invokes the service function. To ensure state continuity, the service function's next input state is always set equal to its most recent output state.

The high-level spec contains a second action, which models the fact that some requests might not be known to the user that invokes the service. For example, if the service is used to redeem cryptographically signed tokens, the user may have incomplete and time-varying knowledge about the set of valid tokens. The high-level spec asserts that the AdvanceState action may only process requests that are *available*, and the action MakeRequestAvailable sets this attribute for a particular request. This action might, for example, correspond to an out-of-band transaction in which the user pays money in exchange for a signed token, thereby enabling the user to submit a request containing that token.

We define two low-level specs, one for Memoir-Basic and one for Memoir-Opt. The Memoir-Basic spec contains variables that model the disk, the RAM, and the TPM's NVRAM. There are seven actions: The MakeOperationAvailable action is an abstraction that directly corresponds to a MakeRequestAvailable action in the high-level spec. The PerformOperation and RepeatOperation actions read and write the RAM and NVRAM in the manner described in §III-B. ReadDisk and WriteDisk exchange data between the disk and the RAM. The Restart action clears the RAM.

The seventh action is CorruptRAM. This models a malicious user's ability to put *nearly* arbitrary values in the RAM before invoking Memoir. The only restriction on the RAM values is that any authenticator in the RAM must be either (1) an authenticator that Memoir has previously returned to the user or (2) a MAC generated with a symmetric key other than the key stored in Memoir's NVRAM.

The Memoir-Opt spec is similar to the Memoir-Basic spec, with an additional variable to model a PCR. The spec contains the seven actions as above, modified to reflect the design described in Section III-C. As a minor example, the Restart action not only clears the RAM but also resets the PCR. This spec also adds two new actions: The TakeCheckpoint action models the checkpoint routine, and the CorruptPCR action models an attacker's ability to extend the PCR.

We prove that the Memoir-Basic spec implements the high-level spec: Any sequence of actions in the Memoir-Basic spec corresponds to a legal sequence of actions in the high-level spec. In particular, MakeOperationAvailable corresponds to MakeRequestAvailable; PerformOperation corresponds to AdvanceState; and the remaining five low-level actions have no effect on the high-level variables. Finally, we prove that the Memoir-Opt spec implements the Memoir-Basic spec, which transitively implies that it implements the high-level spec.

# V. Implementation

In this section, we describe our implementation of Memoir (§V-A) and the modules that use it (§V-B).

## A. Memoir

We have fully implemented both Memoir-Basic and Memoir-Opt, though not the extensions discussed in §III-D and §III-E, nor the setup of the Memoir-Opt checkpoint routine as an SMI handler. Not counting cryptography code we borrowed, this took 1,055 lines of C for the Memoir tools and 1,831 lines of C for MemoirLib, as measured by SLOCCount [39]. We have also implemented variants of Memoir that turn off various features, largely for use in benchmarking. These variants are enabled by compiling MemoirLib with different flags. Thus, one can turn off all state protection, including confidentiality, integrity, and freshness. One can also turn off only freshness protection. And, independently, one can turn off the automatic provisioning of the asymmetric key pair, a setting useful for services that do not need remote verification.

We built Memoir using the Flicker protection framework [24], since it has the smallest available TCB. We made a few modifications to Flicker to make Memoir and our applications work. (1) We increased the space allocated for input and output from 4 KB to 120 KB each, to accommodate large state snapshots. (2) We made Flicker clear the TS bit of CR0 before launching the protected module, so that the module can use floating-point instructions. (3) We built a Flicker simulator, using 1,169 lines of C++, that runs in Windows and allows debugging via Visual Studio. This was of great help, since typically debugging in Flicker requires a long iterated-`printf` approach. Note that all evaluations in §VI run in the real Flicker framework, not the simulated environment.

For cryptographic operations, we used existing C code: Lutz Jänicke's PRNGD and SHA, the PolarSSL implementation of RSA, and Vincent Rijmen's AES. IBM's software TPM implementation [14] was invaluable for debugging purposes. We use a 4-KB PRNGD state size, 1024-bit RSA keys, 128-bit AES keys, SHA-1 hashes, and HMAC-SHA1. We would have preferred to use a more secure hash function, but the TPM only supports SHA-1 [37]. Fortunately, the next version of the TPM specification plans to use other, more secure algorithms [38].

We found the TPM's random number generation somewhat slow, so we implemented the following optimization to avoid the need to use it on every request. Normally, encrypting a state snapshot requires generating a random initialization vector (IV). However, to save time, we instead produce the IV using the PRNG, whose state afterward becomes part of the snapshot. This is safe because the PRNG output is unpredictable to the untrusted system, and because we never encrypt two different state snapshots with the same IV. This latter property stems from the fact that Memoir

makes the system progress through a deterministic series of states, and from the fact that the PRNG part of the state advances each time to produce the IV.

As another optimization, we permit a service to serialize its state into two buffers, one for private data and one for public data. MemoirLib does not encrypt the public data, though it does include that data when computing the authenticator.

### B. Modules

Our `TrInc` module creates, deletes, and updates virtual monotonic counters. It implements TrInc, which is designed to prevent equivocation in distributed systems [20]. After updating a counter, `TrInc` returns a signed certificate associating the counter ID and the old and new values of the counter with a message hash given in the request. Since `TrInc`'s counters are monotonic, certificates cannot associate the counter's transition from $n-1$ to $n$ with two different message hashes. This is useful for preventing equivocation, since it stops a faulty node from sending different messages with the same sequence number $n$ to different nodes. Our `TrInc` module is simpler than TrInc as proposed [20], in two ways. It needs no recent attestation queue to prevent lost outputs since Memoir prevents those as a side effect of preventing lost snapshots. It also lacks symmetric-key support because it is not as useful in our context: Memoir lets the CPU generate signatures rather than slow trusted hardware.

`DiffPriv` calculates differentially private statistics [11] over an encrypted database that untrusted code cannot read. The untrusted code stores the potentially large database on disk and feeds encrypted records to the trusted module, which performs an aggregate calculation over the blinded records. `DiffPriv` assumes that each database record has a timestamp, a unique identifier, and an integer data field; in a real database, this might be a patient's white blood cell count or a user's interest in a set of goods. To query the database, untrusted code specifies the timestamp range over which to query, the operation to calculate over the tuples in that range, and the number of privacy tokens [11] to spend. `DiffPriv` associates each database with a privacy budget, and once the untrusted code has exhausted this budget, it can no longer query the database. This prevents an attacker from issuing many queries and using statistical techniques to remove the noise from the query results. Currently, `DiffPriv` only provides the `average()` aggregator, but implementing more complex statistics is straightforward.

`PassProtect` is a module for safeguarding password-protected data. At initialization, it is passed a secret blob, a low-entropy password, and a high-entropy password. Subsequently, untrusted code accesses the blob by submitting the low-entropy password to `PassProtect`. After three incorrect submissions, `PassProtect` will only release the blob upon reception of the high-entropy password. After the correct high-entropy password is submitted, `PassProtect` allows the low-entropy password to release the blob again. This multi-tier authentication scheme is similar to the PIN+recovery password model that BitLocker [25] uses to protect disk encryption keys.

In all three modules, a successful rollback attack is devastating. If an attacker can roll back `TrInc`, he can generate multiple attested messages with the same counter value, allowing him to equivocate. By rolling back `DiffPriv`, an attacker can arbitrarily respend his privacy budget and gather enough data to remove the statistical noise that hides individual records. By rolling back `PassProtect`, an adversary can launch a brute-force attack on the low-entropy password.

To aid benchmarking, we built a `Noop` module that does nothing but has 1 KB of service state it treats as private.

| Code | Line count |
|---|---|
| Memoir tools | 3,874 |
| Memoir tools, excluding crypto | 1,055 |
| MemoirLib | 6,526 |
| MemoirLib, excluding crypto | 1,831 |

Table I
MEMOIR CODE SIZE, MEASURED BY SLOCCOUNT [39]

## VI. EVALUATION

In this section, we evaluate our Memoir implementation on Linux, demonstrating that it provides fast, secure persistence for trusted modules. First, we show that Memoir's trusted computing base is similar in size to Flicker's TCB, even though Memoir provides much stronger correctness guarantees. We then run microbenchmarks on a variety of popular TPMs, quantifying the slowness of trusted hardware operations and motivating the Memoir optimizations described in §III-C. Finally, we present the results of experiments on our Memoir implementation. These experiments show the overhead of various Memoir features, the time to perform various Memoir operations, and the end-to-end performance seen by applications using Memoir-protected modules.

We perform the Memoir experiments on an HP Compaq 6005 Pro PC running Linux 2.6.31. This system has a 3.0 GHz AMD Athlon II X2 CPU, 2.0 GB of memory, a 160 GB SATA disk, and an Infineon TPM. Since Flicker does not support multiprocessors, we disable all but one of the processors.

### A. Size of Trusted Computing Base

Table I shows the line counts for different parts of Memoir, as measured by SLOCCount [39]. Note that the MemoirLib code base is shared between Memoir-Basic and Memoir-Opt, using compile-time macros to create one or the other library. The table shows that the Memoir tools are only

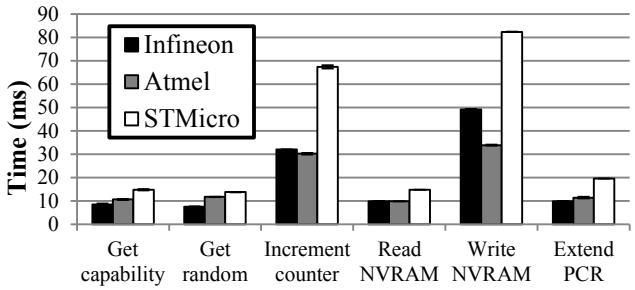


Figure 9. Mean time for various TPM operations. Error bars represent 95% confidence intervals.

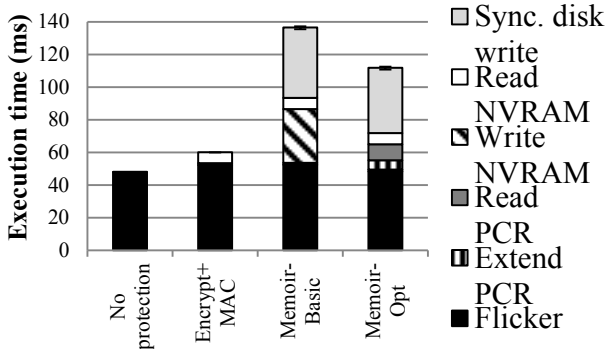


Figure 10. Breakdown of mean request execution time for the `Noop` module. Legend excludes parts so short as to be invisible. Error bars, some of which are too small to see, represent 95% confidence intervals.

3,874 lines, of which all but 1,055 are well-tested, borrowed crypto code. This code runs as root but not within a protected module, so bugs here can deny service to modules but not undermine their protection. The MemoirLib code base, in contrast, is part of the TCB for a protected module. It has 6,526 lines, all but 1,831 of which are well-tested, borrowed crypto code. The small size of MemoirLib indicates that our design and implementation are consistent with our goal of protection through a minimal TCB.

### B. Hardware Microbenchmarks

To better understand the performance implications of using various TPM operations, we performed a series of microbenchmarks on three TPMs: an Infineon v1.2 rev 3.16, an Atmel v1.2 rev 15.4, and an STMicro v1.2 rev 4.30. We extended the jTSS suite [36] to include support for the TPM's NVRAM, monotonic counters, and random number generator. We also instrumented the TPM kernel driver to collect timing information for each command sent to the TPM.

Figure 9 summarizes our results from 100 trials. Writing NVRAM takes 3–6 times as long as reading it. Even the TPM with the slowest write time (the STMicro at 82.4ms), would exhaust its expected 100k writes in less than three hours, if an application wrote as fast as it could. Incrementing a monotonic counter is also relatively slow (30.2–67.5ms), and our measurements indicate that the Infineon rate-limits increments to one every 3.55 seconds, while the Atmel limits increments to one every 4.12 seconds. Contrary to the TPM specification [37], the STMicro does not appear to impose any rate limit, running the risk of the counter rolling over or simply ceasing to function when it reaches its maximum value. To better show the relative performance of these operations, Figure 9 omits the times for Quote (756.0ms, 791.9ms, and 362.3ms for the Infineon, Atmel, and STMicro, respectively). Generating a quote entails a 2048-bit RSA operation on a resource-impoverished device, so its cost is unsurprising.

### C. Effect of Memoir Features on Request Execution Time

In this subsection, we measure the effect of various features Memoir offers. To measure the overhead of rollback resistance, we compare Memoir-Basic to a stripped-down version that provides confidentiality and integrity but not rollback resistance. To measure the overhead of providing confidentiality and integrity, we compare to a minimal version that provides no state protection at all. To measure the savings from our PCR-based performance optimization, we compare to Memoir-Opt.

In all experiments, we perform 1000 request executions of the `noop` module configured with 1 KB of service state. We used the `rdtsc` instruction to record where time is spent.

Figure 10 shows the results. Without any state protection, the cost is essentially just the inherent cost of launching a protection module in Flicker, about 50 ms. Confidentiality and integrity protection costs an additional 7 ms, mostly due to the cost of reading the symmetric key out of NVRAM. Rollback resistance costs an additional 76 ms beyond that, essentially all due to synchronously writing the request to disk (43 ms) before invoking the module and writing NVRAM to store the 20-byte history summary (33 ms). Finally, Memoir-Opt saves 17 ms compared to Memoir-Basic, because it avoids the NVRAM write but incurs an additional 16 ms to read and extend the PCR.

Another difference between Memoir-Basic and Memoir-Opt is that the former uses up the limited write cycles of the NVRAM far more rapidly. The best way to evaluate this would be to compare them on the basis of average time to computer failure, but we lacked the time and resources to conduct such an experiment.

A large part of request execution time in Memoir is the time to synchronously write the request to disk. This time could be substantially reduced if we instead used a solid-state disk or USB flash drive. This would involve NVRAM writes, but to untrusted, inexpensive, high-capacity storage rather than the TPM's trusted, precious, low-performance NVRAM.

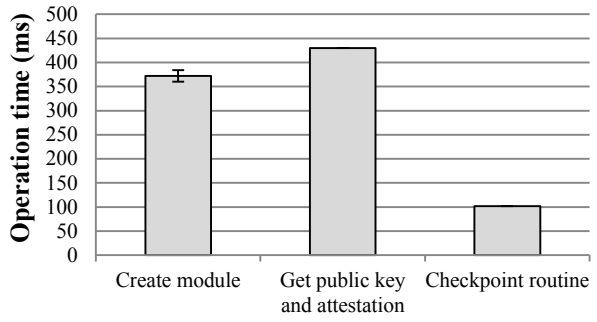


Figure 11. Mean time for various operations, evaluated with the `Noop` module and Memoir-Opt. Error bars, some of which are too small to see, represent 95% confidence intervals.

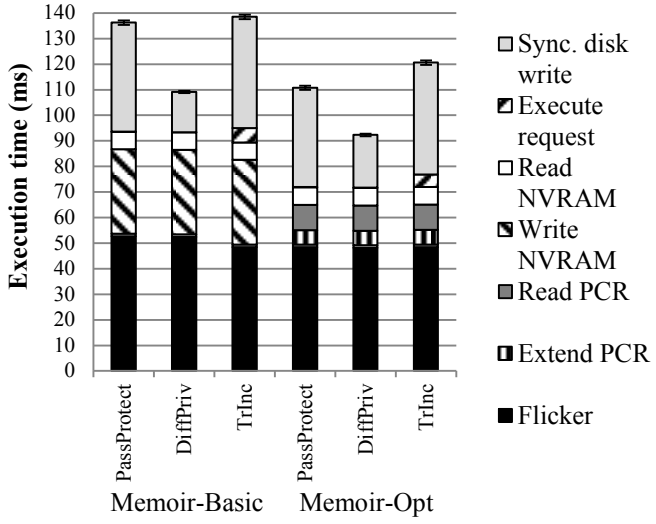


Figure 12. Breakdown of mean request execution time for each of our modules, using Memoir-Basic or Memoir-Opt. Legend excludes parts so short as to be invisible. Error bars represent 95% confidence intervals.

### D. Other Memoir Operations

Next, we evaluate how long it takes to perform other Memoir operations besides request execution: creating a module, obtaining a module's attested public key, and invoking the checkpoint routine in anticipation of shutdown. These operations are less performance critical, since they happen infrequently. For each experiment, we run 1000 trials using the `Noop` module with Memoir-Opt.

Figure 11 shows the results. Creating a module takes the most time, 372 ms, because it involves an expensive RSA key generation. Obtaining an attested public key takes 430 ms since it not only needs to invoke Flicker, it must also request a quote from the TPM. Finally, the checkpoint routine takes 102 ms, mostly to invoke Flicker, read NVRAM and the PCR, and update the NVRAM.

### E. End-to-End Application Time

Finally, we illustrate the end-to-end performance applications can expect from protected modules in Memoir. For this, we install our three application modules, `TrInc`, `DiffPriv`, and `PassProtect`, and evaluate how long it takes them to execute a single request with Memoir-Basic or Memoir-Opt. For `TrInc`, the request is to increment a counter and produce an attestation; for `DiffPriv`, it is to incorporate a private datum into a running aggregate query; for `PassProtect`, it is to verify a correct low-entropy password. We perform 1000 request executions for each module on each of Memoir-Basic and Memoir-Opt.

Figure 12 shows the results. We see that the main differences among modules are due to two factors: synchronous disk writing and request execution. The synchronous disk write takes different amounts of time for each module because each writes a different amount of data to a different location on the disk. Request execution takes about 5 ms in `TrInc`, but negligible time for each of the other modules. This is as expected, since `TrInc` performs a public-key signature operation for each request. Note, though, that this time is significantly less than would be incurred to generate the signature on the TPM itself. This demonstrates the utility of running protected modules on the CPU.

## VII. Related Work

A variety of protected module architectures have been proposed, though none explicitly seek to achieve state continuity. Hence, they seldom include properties like rollback resistance and crash resilience. Nonetheless, these solutions are compelling isolation mechanisms that could complement our techniques for preserving state continuity.

### A. Large Trusted Computing Bases (TCBs)

A number of previous architectures include considerable device driver and other systems code in the TCB. For example, many depend on a full commercial Virtual Machine Monitor (VMM) and its respective management OS (e.g., domain 0 for Xen), which is typically based on a monolithic kernel. Even in slimmer designs, device driver software generally remains in the Trusted Computing Base, since even if the driver code runs at a reduced privilege level, it still handles data on which protected modules' security properties depend. There is also generally no facility for allowing a module to avoid trusting drivers it does not use if other parts of the system require those drivers. Driver code is often the buggiest code in the system [34], and including it still leaves open the possibility of rollback attacks (see §III-E).

KVM/370 [13] and the VAX VMM Security Kernel [17] were two early examples of protected modules that suffer from these limitations. These systems both included block device support in the TCB. More recent examples of systems with these drawbacks include Enforcer [22], Terra [12], Proxos [35], and Overshadow [8].

### B. Tiny Trusted Computing Bases (TCBs)

Other researchers have investigated approaches to minimizing the TCB for protected modules.

Singaravelu et al. develop the Nizza security architecture based on the L4 microkernel [33]. Nizza is able to protect security-sensitive AppCores by running them directly on L4, while the remainder of the legacy software environment runs in a sandbox VM. This is also a compelling architecture, but it does not include any mechanisms for protecting application state against rollback attacks.

TrustVisor [23] and P-MAPS [31] are special-purpose hypervisors that can be loaded on demand. They support a single legacy guest environment, and exist primarily to offer the ability to protect sensitive application code. While these architectures offer a compelling foundation, they do not explicitly include any rollback-resistance mechanisms to protect the nonvolatile state of sensitive code.

Applications built on Flicker [24] or TrustVisor [23] can leverage access to the TPM's monotonic counter for rollback resistance, but the risk of a crash remains. Nizza [33] and P-MAPS [31] also do not explicitly treat the issue of crash resilience for the state of sensitive code, and thus based on the information we have available we conclude they also suffer from a lack of crash resilience.

### C. Applications of Protected Modules

Attested, Append-Only Memory (A2M) is a proposal for a trusted log facility that can be used to prevent adversaries from equivocating (lying differently to different parties) in a distributed system, thereby greatly improving efficiency [9]. The TrInc [20] system improves A2M by showing that a trusted counter on each node is sufficient to detect equivocation. Our framework enables a scalable, performant implementation of TrInc on today's TPM-equipped platforms.

Sarmenta et al. show how to achieve multiple virtual monotonic counters using TPMs available today, without trusting any OS code [32]. Since their TPM-based solutions have poor performance and are unable to offer support for count-limited objects (e.g., n-time use keys), they propose significant changes to the TPM. We show that by adding a small amount of trusted code, we can achieve secure, efficient versions of their protocols using only hardware features available in today's TPMs.

Berger et al. discuss some of the challenges involved in virtualizing the TPM [6]. Their solutions rely on a large VMM (or an expensive secure co-processor), and they do not consider the problem of state continuity.

Katzenbeisser et al. propose extensions to the TPM specification to support revocation of TPM keys [18]. With current TPMs, an attacker who acquires both the access rights to a key and the encrypted version of the key can employ the key in perpetuity. They propose both whitelist- and blacklist-based solutions, but both incur overhead linear in the number of keys on the respective list, and they require changes to the TPM chip. Memoir could provide similar properties with constant time and space overhead and does not require hardware changes.

## VIII. Conclusion

We have presented Memoir, a system for assuring state continuity for protected modules. By tagging each snapshot that leaves the protection boundary with its history summary, Memoir can determine whether a snapshot read back from the untrusted external world is still fresh. Tagging a snapshot with a history summary enables safe identification and replay of previous requests, and hence crash resilience. Our Memoir-Opt design achieves state continuity with little performance overhead on modern hardware, and we propose additional techniques for even better performance via larger-TCB protection frameworks or improved hardware design. We also describe how to support an unlimited number of modules with a constant amount of trusted memory. We formally prove the correctness of our protocols, and demonstrate the utility of our approach by building three applications that demand state continuity for security. Memoir made these protected modules easy to build, by freeing their developers from dealing with issues of assuring state continuity.

## Acknowledgements

This paper benefited from discussions with and suggestions from Helen Wang and David Molnar, as well as from the comments of the anonymous reviewers.

## References

[1] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb. Security analysis and enhancements of computer operating systems. Technical report, Institute for Computer Sciences and Technology, National Bureau of Standards, US Department of Commerce, Apr. 1976.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4), Oct. 1985.

[3] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. HyperSentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of ACM CCS*, 2010.

[4] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, Mar. 2010.

[5] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *OSDI*, 2010.

[6] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the trusted platform module. In *Proceedings of USENIX Security Symposium*, 2006.

[7] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA+ proof system. In *IJCAR*, pages 142–148, July 2010.

[8] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of ASPLOS*, Mar. 2008.

[9] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of SOSP*, 2007.

[10] J. R. Douceur, J. R. Lorch, B. Parno, J. Mickens, and J. M. McCune. Memoir—formal specs and correctness proofs. Technical Report MSR-TR-2011-19, Microsoft Research, Feb. 2011.

[11] C. Dwork. Differential privacy. In *Proceedings of ICALP*, July 2006.

[12] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SOSP*, 2003.

[13] B. D. Gold, R. R. Linde, and P. F. Cudney. KVM/370 in retrospect. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1984.

[14] K. Goldman. IBM's software trusted platform module. http://ibmswtpm.sourceforge.net/.

[15] T. Hardjono and G. Kazmierczak. Overview of the TPM key management standard. TCG Presentations: https://www.trustedcomputinggroup.org/news/, Sept. 2008.

[16] Intel Corporation. Intel architecture software developer's manual, Oct. 2005.

[17] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, Nov. 1991.

[18] S. Katzenbeisser, K. Kursawe, and F. Stumpf. Revocation of TPM keys. In *Proceedings of the Conference on Trust and Trustworthy Computing*, Apr. 2009.

[19] L. Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Professional, 2002.

[20] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proc. of USENIX NSDI*, 2009.

[21] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell. The SMART way to migrate replicated stateful services. In *EuroSys*, Apr. 2006.

[22] J. Marchesini, S. W. Smith, O. Wild, J. Stabiner, and A. Barsamian. Open-source applications of TCPA hardware. In *Proceedings of IEEE ACSAC*, 2004.

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

[24] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, Apr. 2008.

[25] Microsoft Corporation. BitLocker drive encryption. http://technet.microsoft.com/en-us/library/dd548341%28WS.10%29.aspx.

[26] S. C. Misra and V. C. Bhavsar. Relationships between selected software measures and latent bug-density. In *Proceedings of the Conference on Computational Science and Its Applications*, Jan. 2003.

[27] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ASPLOS*, Mar. 2009.

[28] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[29] R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, 1978.

[30] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *SOSP*, 2001.

[31] R. Sahita, U. Warrier, and P. Dewan. Dynamic software application protection. http://blogs.intel.com/research/trusted%20dynamic%20launch-flyer-rls_pss001.pdf, Apr. 2009.

[32] L. Sarmenta, M. van Dijk, C. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In *Proceedings of the ACM Workshop on Scalable Trusted Computing (STC)*, 2006.

[33] L. Singaravelu, C. Pu, H. Haertig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *EuroSys*, 2006.

[34] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 22(4), 2004.

[35] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of OSDI*, 2006.

[36] R. Toegl, T. Winkler, M. E. Steurer, M. Pirker, C. Pointner, T. Holzmann, M. Gissing, and J. Sabongui. IAIK jTSS - TCG software stack for the Java platform. v0.5 http://trustedjava.sourceforge.net/.

[37] Trusted Computing Group. Trusted Platform Module Main Specification. Version 1.2, Revision 103, 2007.

[38] Trusted Computing Group. Summary of Features Under Consideration for the Next Generation of TPM. http://www.trustedcomputinggroup.org, 2009.

[39] D. A. Wheeler. Linux kernel 2.6: It's worth more! http://www.dwheeler.com/essays/linux-kernel-cost.html, Oct. 2004.

[40] A. C. Yao. Theory and applications of trapdoor functions. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, 1982.