Provably-Secure Remote Memory Attestation for Heap Overflow Protection

Alexandra Boldyreva[1], Taesoo Kim[1], Richard Lipton[1], and Bogdan Warinschi[2]

[1] Georgia Institute of Technology
[2] University of Bristol

Abstract. Memory corruption attacks may lead to complete takeover of systems. There are numerous works offering protection mechanisms for this important problem. But the security guarantees that are offered by most works are only heuristic and, furthermore, most solutions are designed for protecting the local memory. In this paper we initiate the study of *provably secure* remote memory attestation; we concentrate on provably detecting heap-based overflow attacks and consider the setting where we aim to protect the memory in a remote system. We present two protocols offering various efficiency and security trade-offs (but all solutions are efficient enough for practical use as our implementation shows) that detect the presence of injected malicious code or data in remotely-stored heap memory. While our solutions offer protection only against a specific class of attacks, our novel formalization of threat models is general enough to cover a wide range of attacks and settings.

1 Introduction

Memory corruption attacks are among the most common techniques used to take control of arbitrary programs. These attacks allow an adversary to exploit running programs either by injecting their own code or diverting program's execution, often giving the adversary complete control over the compromised program. While this class of exploits is classically embodied in the buffer overflow attack, many other instantiations exist, including use-after-free vulnerabilities and heap overflow. The latter is the focus of our work. Without question, this problem is of great importance and has been extensively studied by the security community.

Existing solutions (such as Stack and heap canaries [14, 17, 19, 31] or address space layout randomization (ASLR) [27, 37], etc.) vary greatly in terms of security guarantees, performance, utilized resources (software or hardware-based), etc. While these techniques are implemented and deployed in many systems to prevent a number of attacks in practice, their constructions are only appropriate in the context of local systems: for example, an authority checking the integrity of heap canaries, has to monitor every single step of the program's execution. However, this requirement is making the existing heap-based protection schemes hardly applicable to remote memory attestation where the authority might reside outside of a local machine. For example, a straight-forward construction to keep track of all locations of heap canaries and validate their integrity upon request not only incurs noticeable performance overheads, but also requires a trusted communication channel between the program and a remote verifier.

More critically, none of the prior works targeting heap overflow attacks provided provable security guarantees. Without a clear adversarial model it is hard to judge the scope of the protection, and often the attackers, who are getting more and more sophisticated, are still able to bypass many such mitigation techniques.

Proving that a given protocol can resist all possible attacks within a certain well-defined class is the gold standard in modern cryptography. However, protocols that are provably secure are rather rarely used in real systems either because they commonly target extremely strong security definitions and hence are too slow for practical use, or they rely on impractical assumptions about attackers. Our work tries to bridge this gap in the context of remote attestation by designing practical protocols with provable security guarantees against realistic threats and satisfying practical system requirements. Our treatment utilizes the formal provable-security approach of modern cryptography that works hand in hand with applied systems expertise.

In this paper, we realized our theoretical findings as a working prototype system that can mitigate, (still limited), heap overflow attacks in applications running remotely outside of user's local computer. Although the current implementation therefore focuses on protecting user's programs running on the cloud environment or firmware running outside of the main CPU, the proposed security model is general enough to be useful for future works addressing other classes of adversaries. We now discuss our focus and contributions in more detail.

OUR FOCUS. Our focus is on the *remote* verification setting, motivated by the widespread use of cloud computing. In our setting, two entities participate in the protocol; a program that is potentially vulnerable, and a remote verifier who attests the state of the program's memory (e.g., heap). This setting is particularly useful for verifying the integrity of software that is deployed and runs outside of a local machine: a deployed program on the cloud is one example, and a firmware running outside of the main CPU is another example. Note that if the cloud is completely untrusted, we cannot guarantee security without relying on secure hardware (and our focus is software-based solution only). Hence we need to trust the cloud to a certain degree, but at the same time we want to avoid changing the operating system there. Since we do not trust the program which is potentially malicious, we create another entity, a wrapper, that is not directly affected by the program, unless an adversary bypasses the protection boundary provided by an operating system.

In practice, system software (e.g., browser or operating system) is vulnerable to memory corruptions because it heavily relies on unsafe low-level programming languages like C for either performance or compatibility reasons. As we mentioned, we do not attempt to prevent entire classes of memory corruption attacks (e.g., use-after-free or bad-casting) nor exploitation techniques (e.g., return-oriented programming (ROP)) with one system. We only consider one particular type of memory corruption attack that overwrites a consecutive region of memory (e.g., buffer) to compromise a control-sensitive data structure (e.g., function pointer or virtual function table). However, we believe such memory corruptions are still very common (e.g., the recent GHOST vulnerability in GLibc [4]), and become more important in the cloud setting where we have to rely on the cloud provider.

Within this scope, our goal is to find solutions that (1) provide provable security guarantees and (2) are practically efficient.

RMA SECURITY DEFINITION. Providing security guarantees is not possible without having a well-defined security model. We start with defining a *remote memory attestation (RMA)* protocol, whose goal is protecting the integrity of a program's data memory (e.g., heap). It is basically an interactive challenge-response protocol between a prover and a verifier, which is initialized by a setup algorithm that embeds a secret known to the verifier into a program's memory. The goal of the verifier is to detect memory corruptions.

Next we propose the first security model for RMA protocols. The definition is one of our main contributions. Our model captures various adversarial capabilities (what attackers know and can do), reflecting real security threats. We assume that an attacker can have some a-priori knowledge of the memory's contents (e.g., binary itself) and can learn parts of it, adaptively, over time.

Since we target a setting where the communication between the prover and the verifier is over untrusted channels, we let the adversary observe the legitimate communication between the prover and the verifier. Moreover, we let it impersonate either party and assume it can modify or substitute their messages with those of its choice. To model malicious writes to the memory we allow the attacker to tamper the memory. The goal of the attacker is to make the verifier accept at a point where the memory is corrupted.

We note that on the one hand no security may be possible if an attacker's queries are unrestricted and on the other we would like to avoid hardwiring in the model a particular set of restrictions on these queries. Accordingly we state security with respect to abstract classes of functions that model the read and write capabilities of the attackers. This allows us to keep the definition very general. We leave it for the theorem statements that state the security of particular protocols to specify these classes, and thus define the scope of attacks the protocol defends against.

To prevent against the aforementioned GHOST attack [4] where a read (e.g., information leak) follows by write to the same location and leaves the key intact, any solution in our setting needs to perform a periodic key refresh. Our protocol definition and the security model take this into account. But of course, we do not guarantee security if the attack happens within a refresh time window. This is a common caveat with preventing timing attacks.

An RMA protocol proven to satisfy our security definition for specific read and write capabilities classes would guaranty security against *any* efficient attacker with such practical restrictions, under reasonable computational assumptions. This is in contrast to previous schemes, which were only argued to protect against certain specific attacks, informally.

PROVABLY-SECURE RMA CONSTRUCTIONS. The idea underlying our solutions is simple and resembles the one behind stack or heap canaries. We embed secrets throughout the memory and, for attestation, we verify that they are intact. This is similar to how canaries are used, but for the setting where the verifier is remote the ideas need to be adapted. A simple but illustrative example is the protocol where the prover simply sends to the verifier the hash of all of the (concatenated) canaries. Here, the attacker can replay this value after modifying the memory. The following discussion illustrates further potential weaknesses of this protocol uncovered when trying to derive provable security guarantees.

For clarity, instead of calling the secrets canaries, let us refer to the secrets we embed in the memory as shares, i.e., we split a secret into multiple shares and spread them out in memory. For now, let's assume for simplicity that the shares are embedded at equal intervals. Then an adversary who injects malicious code, and hence writes a string that is at least one-block long, will over-write at least one share, even if it knows the shares' locations. Verification just checks whether the original secret can be reconstructed and used in a simple challenge-response protocol that prevents re-plays. For example, the verifier could send a random challenge, and the prover would reply with the hash of the reconstructed secret and the challenge. Note that the prover will run in a totally separate memory space so the secrecy of the reconstructed key at time of verification is not an issue.

We note that our solution does not readily apply for the stack because the stack doesn't have explicit unit or boundaries to statically place secret shares), unlike the heap that has a unit (a page) of allocation that makes the key placement efficient and easy.

The standard security of an n-out-of-n secret sharing scheme ensures that unless the attacker reads all memory (and in this case no security can be ensured anyway), the key is information-theoretically hidden. However, the adversary could read and then tamper the memory while leaving the share intact. To mitigate this, the periodic updates could re-randomize all shares, while keeping the same secret. The size of the blocks and the frequency of the updates are the parameters that particular applications could choose for the required tradeoff between security and efficiency. In the ideal setting, we would refresh the shares whenever the leakage of a share happens. However, since the occurrence of such events is not always clear, the alternative solution of refreshing “often" enough may lead to unreasonable overheads. In our current implementation, we keep the frequency of updates a parameter and developers can simply incorporate timing that reflect realistic assumptions on the adversary in our implementation.

Although the solution approach seems simple and sound, it turns out that assessing its security and practicality raises numerous subtleties and complications, both from the systems and cryptographic points of view. For example, our system can not fix the size of memory object, which naturally underutilizes the memory space (e.g., de-fragmentation). In our system, we support various memory slots for allocation, from the smallest 8 byte objects incrementally to over 100 mega bytes, depending on the user's configuration.

The obvious choice for producing the secrets to be embedded in the memory is to use an n-out-of-n secret sharing scheme as a building block for our constructions. It turns out however, that the standard security of secret sharing schemes is not sufficient to guarantee the security of the protocol. First, we have to extend the security definition to take into account key updates. The attacker should be able to access the whole memory as long as it does not do it in between consecutive updates. The extended notion is known as proactive secret sharing [20]. Also, for the proof we need the additional properties that modifying at least one share implies changing a secret, and one extra property we discuss later. Fortunately, all these properties are satisfied by a simple XOR-based secret sharing scheme.

We show that combing the simple XOR-based secret sharing scheme (or any generic secret sharing scheme with some extra properties we define) and the hash-based challenge-response protocol yields a secure and efficient RMA protocol, for attackers with restricted, but quite reasonable abilities to read and tamper the memory. The proof we provide relies on the random oracle (RO) model [8]. Since the RO is unsound [12] for security-critical applications it may be desirable to have protocols which provably provide guarantees in the standard (RO devoid) model.

An intuitively appealing solution is to employ some symmetric-key identification protocol, e.g., replying with a message authentication code (MAC) of the random challenge, where the MAC is keyed with the reconstructed secret. However, given the capabilities that we ascribe to realistic adversaries, a formal proof would require a MAC secure in the presence of some leakage on and tampering of the secret key. The latter property is also known as security against related key attacks (RKA) [6]. Unfortunately, there are no suitable leakage and tamper-resilient MACs for a wide class of leakage and tampering functions, as the existing solutions, e.g. [5, 10], only address specific algebraic classes of tampering functions and are rather inefficient.

Perhaps unexpectedly, we consider a challenge response protocol based on a public key encryption scheme – the verifier sends a random challenge and expects an encryption of the challenge together with the (reconstructed) secret. This solution requires that the public key of the verifier is stored so that it is accessible by the prover, and cannot be tampered (otherwise we would need a public-key scheme secure with respect to related public key attacks, and similarly to the symmetric setting, there are no provably secure schemes wrt this property, except for few works addressing a narrow class of tamper functions [7, 38]).

To ensure non-malleability of the public key, our system separates the memory space of a potentially malicious program from its prover (e.g., different processes), and store its public key in the prover's memory space. Since the verification procedure is unidirectional (e.g., a prover accesses the program's memory), our system can guarantee the non-malleability of the public key in practice (e.g., unless no remote memory overwriting or privilege escalation). This level of security is afforded by deployed computational platforms (e.g. MMU commodity processors).

It is natural to expect some form of non-malleability from the encryption scheme. Otherwise, the attacker could modify a legitimate response for one challenge into another valid one for the same key and a new challenge. An IND-CCA secure encryption such as Cramer Shoup [15] could work for us. We note however that IND-CCA secure is an overkill for our application since we do not need to protect against arbitrary maulings of the ciphertext; instead, the attacker only needs to produce a valid ciphertext for a particular message, known to the verifier. We show that an encryption scheme secure against a weaker notion of plaintext-checking attacks [32] is sufficient for us. Accordingly, we use the "Short" Cramer-Shoup (SCS) scheme proposed and analyzed very recently by Abdalla et al. [1]. This allows us to save communication one group element compared to regular Cramer Shoup. We discuss that one can optimize further and save an additional group element in the communication by slightly increasing computation.

IMPLEMENTATION RESULTS. To demonstrate the feasibility of RMA, we implemented a prototype system that supports arbitrary programs without any modification (e.g., tested with popular software with a large codebase, such as Firefox, Thunderbird and SPEC Benchmark). Our evaluation shows that the prototype incurs very small performance overheads and detects heap-based memory corruptions with the remote verifier.

In a bit more detail, we implemented both, the hash- and encryption-based, protocols. Interestingly, both protocols showed similar performance, despite the latter one relying on public key operations, which are much slower than a hash computation. This is because the significant part of the performance overhead comes from the implementation of the custom memory allocator, side-effects of memory fragmentation and network bandwidth, which all make the differences in times of crypto operations insignificant.

RELATED WORK. The works that is perhaps closest in spirit and application domain to ours is by Francillon et al. [18] who address the problem of remote device attestation. Their approach is also based on provable security, but consider a significantly weaker model where the adversary not tamper or read parts of the internal memory of the device. These are key features of the adversary that we aim to defend against.

Canaries are random values placed throughout a stack or heap, which are later checked by the kernel. Canary-based protection has been adopted to prevent stack smashing [2]: e.g., ProPolice [17], StackGuard [14], StackGhost [19]. Similarly, canaries (or guard as a general form) have been used for heap protection, in particular metadata of heap [33, 39] (e.g., double free): HeapShield [9] or AddressSanitizer [34]. These solutions do not immediately work in our setting. This is mainly because all canaries need to be sent and checked by the remote verifier without leaking or without being compromised by an adversary. While heavy solutions like employing secure channels (e.g. TLS) would help mitigate this problem, the resulting system would need to transfer large quantities of data, making it unsuitable for practical use.

Our solutions could be viewed as a novel variant of "compact" cryptographic canaries, suitable for remote setting and providing provable security guarantees under precisely defined threat models.

Software-based attestation has been explored in various contexts: peripheral firmware [16, 23, 25], embedded devices [13, 24, 36], or legacy software [35]. That line of work, which falls under the generic idea of *software based attestation* is different from ours in two main differences. First, the setting of firmware attestation uses a different adversarial model. There, an adversary aims to tamper with the firmware on a peripheral and still wants to convince an external verifier that the firmware has not been tampered with. In its attack, the adversary has complete access to the device prior to the execution of the attestation protocol; the protocol is executed however without adversarial interference. Our model considers an adversary who can glean only partial information on the state of the memory prior to its attack, but who acts as man-in-the-middle during the attestation protocol.

Challenge-response protocols are natural solutions in both situations. Since we aim for solutions that admit rigorous security proofs we rely on primitives with cryptographic guarantees. In contrast due to constraints imposed by the application domain solutions employed peripheral attestation cannot afford to rely on cryptographic primitives. Instead, constructions employ carefully crafted check-sum functions where unforgeability *heuristically* relies on timing assumptions and lack of storage space on the device. Jacobsson and Johansson [22] show that such assumptions can be grounded in the assumptions that RAM access is faster than access to the secondary storage [22]. Our work is similar in its goals with that of Armknecht et al. [3] who provide formal foundations for the area of software attestation.

More recently, a handful of hardware-based (e.g., coprocessor or trusted chip) attestation has been proposed as well: Flicker [29] and TrustVisor [28] using TPM, and Haven using Intel SGX [21, 30]. Our work differs in that we do not explicitly rely on hardware assumptions and provides provable security guarantee.

Finally, a recent paper [26] addresses the problem of a virus detection from a provable security perspective. The authors introduce the virus detection scheme primitive that can be used to check if computer program has been infected with a virus injecting malicious code. They describe a compiler, which outputs a protected version of the program that can run natively on the same machine. The verification is triggered by an external verifier. Even though the considered problems and the basic idea of spreading the secret shares are similar, the treatment and the results in [26] are quite different from ours. The major difference is that the attacker in the security model of [26] is not allowed to learn any partial information about the secret shares. Our security definition, in turn, does take partial leakage of the secret into account. Their security definition, however, allows the attacker to learn the contents of the registers during the attack. This is not a threat in our setting since the computations happen within the trusted wrapper. Also, their solutions do not rely on the PKI, which is a plus. The other important difference is that the proposal in [26] is mostly of theoretical interest (as they rely on leakage-resilient encryption for which there are no efficient implementations), while our solution is quite efficient. The work [26] has additional results about protection against tiny overwrites but that requires CPU modifications.

2 Notation

$X \xleftarrow{\$} S$ denotes that X is selected uniformly at random from S. If A is a randomized algorithm, then the notation $X \xleftarrow{\$} A$ denotes that X is assigned the outcome of the experiment of running A, possibly on some inputs. If A is deterministic, we drop the dollar sign above the arrow. If X, Y are strings, then $X \| Y$ denotes the concatenation of X and Y. We write $L :: a$ for the list obtained by appending a to the list L and $L[i, \ldots, j]$ for the sublist of L between indexes i and j. We write id for the identity function (the domain is usually clear from the context) and write $\mathcal{U}_S$ for the uniform distribution on set S. If n is an integer we write $[n]$ for the set $1, 2, \ldots, n$. For an integer k, and a bit b, b^k denotes the string consisting of k consecutive "b" bits.

3 Remote Memory Attestation

SYNTAX. We start with defining the abstract functionality of *remote memory attestation (RMA)* protocol.

Definition 1. [RMA protocol] *A remote memory attestation protocol is defined by a tuple of algorithms* $(\mathsf{SS}, \mathsf{Init}, (\mathsf{MA}, \mathsf{MV}), \mathsf{Update}, \mathsf{Extract})$ *where:*

- *The setup algorithm* SS *takes as input a security parameter* 1^κ *and outputs a pair of public/secret keys* (pk, sk)*. (*SS *is run by the verifier.) This output is optional.*
- *The initialization algorithm* Init *takes as input a bitstring* M *(representing the memory to be protected), a public key* pk *and the secret key* sk *and outputs a bitstring* M_s *(that represents the protected memory), and a bitstring* s *(secret information that one can use to certify the state of the memory).*
- *The pair of interactive algorithms* $(\mathsf{MA}, \mathsf{MV})$*, run by the prover and verifier resp., form the attestation protocol. Algorithm* MA *takes as inputs the public key* pk *and a bitstring* M_s *and the verifier takes as inputs the secret key* sk *and secret* s*. The verifier outputs a bit, where 1 indicates acceptance, and 0 – rejection.*
- *The update algorithm* Update *takes as input a bitstring* M_s *and outputs a bitstring* ${M_s}'$ *(this is a "refreshed" protected memory). It can be ran by the prover at any point in the execution.*
- *The* $\mathsf{Extract}$ *algorithm takes as input a bitstring* M_s *(representing a protected memory) and outputs a bitstring* M *(represented the real memory protected in* M_s*) and secret* s*. This is used in the analysis mostly, but also models how the OS can read the memory.*

The correctness condition requires that for every (pk, sk) *output by* SS*, every* $M \in \{0,1\}^*$*, and every* (M_s, s) *output by* $\mathsf{Init}(M, pk, sk)$*, the second party in* $(\mathsf{MA}(pk, M_s), \mathsf{MV}(sk, s))$ *returns 1 with probability 1. Also,* $\mathsf{Extract}(M_s) = (M, s')$ *for some* s' *with probability 1. These conditions should hold even for an arbitrary number of runs of* Update *protocol.*

In practice the remote verifier initializes the wrapper with the secret before being sent to the cloud. The wrapper later acts as the local prover to the remote verifier. In practice the wrapper is a separate process that gets memory access via *ptrace* mechanism.

RMA SECURITY. We now formally define the security model for an RMA protocol, which is part of our main contributions.

We consider an attacker who can read the public key (if any), and can observe the interactions between the prover and the verifier. The attacker works in two stages. In the first stage of its attack, it can read arbitrary parts of the memory and can over-write a part of the memory by injecting data of its choosing. In this phase, the adversary can observe and interfere with the interaction between the prover and the verifier. This is captured by giving the adversary access to the oracles that execute the interactive RMA protocol; in particular, the adversary can chose to observe a legitimate execution of the protocol by simply forwarding the answers of one oracle to the other. Of course, the adversary can choose to manipulate the conversation, or even supply inputs of its own choosing. We only model a single session of the protocol as we do not expect parallel sessions to be run in practice. Also, at any point, the attacker can request that the shares of the secret get updated. In the second stage the adversary specifies how it wants to alter the memory (where and what data it wants to over-write). The memory is modified, one extra update is performed, and then the attacker can continue its actions allowed in the first stage, with the exception that it is not given the ability to read the memory anymore, and this is the reason we consider two stages of the attacker. This captures the fact noted in the Introduction, that security is only possible if the memory update procedure is performed in between the read and write, which can be arbitrary and thus leave the secret intact (by reading and over-writing it).

We say that the adversary wins if it makes the verifier accept in the second stage, despite the memory being modified by the attacker. This captures the idea that the verifier does not notice that the memory has been corrupted.

We observe that it is necessary to restrict the adversary's abilities, for a couple of reasons. First, as we mentioned in the Introduction, no security may be possible if an attacker's queries are unrestricted. For instance, the adversary may read the whole memory in between the secret updates or it could read a block and immediately over-write it maintaining intact the associated secret share. Moreover, note that an adversary who can over-write memory bit by bit, could eventually learn the whole secret by fixing each bit for both possible values, one by one and then observing the outcome of the interaction between the prover and the verifier. In short, no security is possible if we do not impose (reasonable) restrictions on the adversary.

Second, it seems unlikely that a unique solution suffices to protect against a wide class of attacks and that different solutions would work for different applications and classes of attacks. Yet, we would want to avoid providing a diffrent security definition for each individual scenario.

Accordingly, we state security with respect to abstract classes of functions that parametrize the read and write queries that model the legitimate read and tamper requests the attacker can do. This allows our definition to be quite general; we leave it to the theorem statements for particular protocols and applications to specify these classes and hence clarify the scope of attacks the protocol prevents against.

```
$\mathbf{Exp}^{\mathrm{rma}\text{-}(\mathcal{L},\mathcal{T})}_{A,\Pi}$:
$(pk, sk) \leftarrow \mathsf{SS}$
$M \leftarrow A(pk)$
$(M_s, s) \leftarrow \mathsf{Init}(M, pk, sk)$
$g' \leftarrow A^{\mathsf{Read}(\cdot),\mathsf{Tamper}(\cdot),\mathsf{MA}(pk,M_s),\mathsf{MV}(sk,s),\mathsf{Update}}$
If $g \notin \mathcal{T}$ return $\perp$
$M_s \leftarrow \mathsf{Update}(M_s)$
$M_s \leftarrow g'(M_s)$
$A^{\mathsf{Tamper}(\cdot),\mathsf{MA}(pk,M_s),\mathsf{MV}(sk,s)}$
Output $1$ iff $\mathsf{MV}$ accepts in the 2nd stage
and at that point the first part of $\mathsf{Extract}(M_s)$ is not $M$.

Oracle $\mathsf{Read}(f)$:
if $f \notin \mathcal{L}$ return $\perp$
    otherwise return $f(M_s)$

Oracle $\mathsf{Tamper}(g)$:
if $g \notin \mathcal{T}$ return $\perp$
$M_s \leftarrow g(M_s)$
```

Fig. 1: Game defining the security of the memory attestation scheme $\Pi = (\mathsf{SS}, \mathsf{Init}, (\mathsf{MA}, \mathsf{MV}), \mathsf{Update}, \mathsf{Extract})$.

Definition 2. [RMA scheme security.] *Let $\mathcal{L}$ and $\mathcal{T}$ be two classes of* leakage *and* tampering *functions. Consider an RMA protocol* $\Pi = (\mathsf{SS}, \mathsf{Init}, (\mathsf{MA}, \mathsf{MV}), \mathsf{Update}, \mathsf{Extract})$. *We define its security via the experiment* $\mathbf{Exp}^{\text{rma-}(\mathcal{L},\mathcal{T})}_{A,\Pi}$ *involving the adversary A which we present in Figure 1.*

We call Π secure wrt $\mathcal{L}$ and $\mathcal{T}$ if for every (possibly restricted) efficient adversary A the probability that $\mathbf{Exp}^{\text{rma-}(\mathcal{L},\mathcal{T})}_{A,\Pi}$ *returns 1 is negligible in the security parameter.*

The design of the above model is influenced directly by studying the practical threats. In particular, reading memory to leak information has been a prerequisite pretty much to all attacks from ten years back. Taking the man-in-the-middle attacks into account is motivated by the observation that even though we trust the cloud provider, we do not necessarily trust the path between the provider and the client, e.g., when using a cafe's wifi. We demand that the secure attestation be done without employing secure channels.

REMARK. Turns out that the practical classes of read and write functions may not describe the necessary restrictions by themselves. Thus one can further restrict the adversaries, but again, this is done in the security statements. For instance, security of our constructions will tolerate any attacker who can read all but one "block" of the memory and can over-write any arbitrary part of the memory as long as that part is longer than some minimum number of bits.

4 Building Blocks

REFRESHABLE SECRET SHARING SCHEME. Our schemes rely on an n-out-of-n secret sharing scheme where one needs all of the shares to reconstruct the secret; any subset of $n-1$ shares is independent from the secret. In addition to the standard property, we also require that it is possible to refresh shares in such a way that all subsets of $n-1$ shares, each obtained in between updates, are independent of the secret. This property is known as proactive secret sharing [20]. In addition, we require two more security properties which we describe later in this section.

Syntax. We first provide the syntax of the secret sharing schemes that we consider.

Definition 3. *A* refreshable n-out-of-n secret sharing scheme *is defined by algorithms* $(\mathsf{KS}, \mathsf{KR}, \mathsf{SU})$ *for sharing and reconstructing a secret, and for refreshing the shares*[3]. *For simplicity we assume that the domain of secrets is* $\{0,1\}^\kappa$ *(where κ is the security parameter). The sharing algorithm* KS *takes a secret s and outputs a set* $(s_1, s_2, \ldots, s_n)$ *of shares*[4]. *The reconstruction algorithm* KR *takes as input a set of shares* $s_1, s_2, \ldots, s_n$ *and returns a secret s. The update algorithm* SU *takes as input a set of shares* $(s_1, s_2, \ldots, s_n)$ *and returns the updated set* $(s'_1, s'_2, \ldots, s'_n)$*, a new re-sharing of the same secret.*

[3] We use the mnemonics KS, KR to indicate that we think of the secret as being some cryptographic key.

[4] We do not use the set notation for simplicity.

For correctness we demand that for any $s \in \{0,1\}^\kappa$ and any $(s_1, s_2, \ldots, s_n)$ obtained via $(s_1, s_2, \ldots, s_n) \xleftarrow{\$} \mathsf{KS}(s)$ it holds that $\mathsf{KR}((s_1, s_2, \ldots, s_n)) = s$ and $\mathsf{KR}(\mathsf{SU}^i((s_1, s_2, \ldots, s_n)) = s$ with probability 1 for any integer $i \geq 1$, where $\mathsf{SU}^i((s_1, s_2, \ldots, s_n))$ denotes i consecutive invocations of SU as $\mathsf{SU}(\mathsf{SU}(\ldots \mathsf{SU}((s_1, s_2, \ldots, s_n)) \ldots))$.

Security. We require that the secret sharing scheme that we use satisfies three security properties.

SECRET PRIVACY. The most basic one, secret privacy for refreshable secret sharing scheme (aka proactive secret sharing) guarantees that $n-1$ shares do not give the adversary any information about the secret, and this holds even for an arbitrary number of updates to each set of shares. The formal definition is in the full version [11].

OBLIVIOUS RECONSTRUCTION. We also require that the scheme enjoys *oblivious reconstruction*. Intuitively, this demands that given an adversary who can read and replace some of the shares, it is possible to determine at any point if the value encoded in the shares is the same as the original value or not. This property is related but is different from Verifiable Secret Sharing: the ability to tell that the shares are consistent with some secret does not necessarily mean that one can tell if transforming a set of shares to another (valid) one has changed or not the underlying secret.

More formally, fix a secret $s \in \{0,1\}^\kappa$ and let $(s_1, s_2, \ldots, s_n) \xleftarrow{\$} \mathsf{KS}(s)$. Consider an adversary who can intermitently issue two types of querries. On a query $i \in \{1, \ldots, n\}$ the adversary receives s_i; on a query $(i, v) \in (\{1, 2, \ldots, n\} \times \{0,1\}^\kappa$ the value of s_i is set to v.

We require that there exists a "secret changed?" algorithm SC, formalized in Figure 2, which given the queries made by A and the answers it receives can efficiently decide (with overwhelming probability) if the value of the secret that is encoded is equal to the value of the original secret.

```
$\mathbf{Exp}^{\mathrm{rec}}_{A,\Pi}$:
$s \xleftarrow{\$} A; L \leftarrow [\,]$
$\{s_1, s_2, \ldots, s_n\} \leftarrow \mathsf{KS}(s)$
$A^{\mathsf{ShareInfo}(\cdot)}$
$b \leftarrow SC(L)$
$s' \leftarrow \mathsf{KR}(s_1, s_2, \ldots, s_n)$
return 1 iff
$(b = 1$ and $s = s')$ or $(b = 0$ and $s \neq s')$

Oracle $\mathsf{ShareInfo}(\cdot)$:
On input $i \in \{1, \ldots, n\}$
$L \leftarrow L :: (i, s_i)$
  return $s_i$
On input $(i, v) \in \{1, 2, \ldots, n\} \times \{0,1\}^\kappa$
$L \leftarrow L :: (i, v)$
  $s_i \leftarrow v$
```

Fig. 2: Experiment defining the oblivious reconstruction property for secret sharing.

SHARE UNPREDICTABILITY. This property demands that for any secret (chosen by the adversary) and any sharing of the secret, following an Update an adversary cannot tamper (in any meaningful way) with any of the resulting fresh shares in a way that does not alter the secret. This intuition is formalized using the game $\mathbf{Exp}^{\mathrm{unpred}}_{A,\mathit{\Pi}}$ in Figure 3. First,the experiment samples a random secret. After the adversary learns some (but not all) of the shares, the shares are refreshed, and the adversary needs to tamper with at least one share. The adversary wins if the secret that is shared stayed unchanged through the process. We say that $\mathit{\Pi}$ satisfies share unpredictability if for any adversary which calls the Read oracle at most $n-1$ times and the Tamper oracle at least once, the probability that the experiment returns 1 is negligible.

```
$\mathbf{Exp}^{\mathrm{unpred}}_{A,\mathit{\Pi}}$:
$s \leftarrow \{0,1\}^\kappa$
$A^{\mathsf{Read}(\cdot)}$
$(s_1, s_2, \ldots, s_n) \leftarrow \mathsf{SU}(s_1, s_2, \ldots, s_n)$
$A^{\mathsf{Tamper}(\cdot)}$
return $s \overset{?}{=} \mathsf{KS}(s_1, s_2, \ldots, s_n)$

Oracle $\mathsf{Read}(i)$
    return $s_i$
Oracle $\mathsf{Tamper}(i, v)$
    $s_i \leftarrow v$
```

Fig. 3: Game defining share unpredictability for for secret sharing. We demand that A queries his Tamper at least once.

Secure Construction. Here we present a very simple n-out-of-n refreshable secret-sharing scheme with oblivious reconstructability and argue its security.

Construction 41 [Refreshable secret sharing] We define the scheme $(\mathsf{KS}, \mathsf{KR}, \mathsf{SU})$ as follows.

- KS takes secret $s \in \{0,1\}^\kappa$, picks $s_i \overset{\$}{\leftarrow} \{0,1\}^\kappa$ for $1 \leq i \leq n-1$, computes $s_n \leftarrow s \oplus s_1 \oplus \ldots \oplus s_{n-1}$
- KR on input $(s_1, \ldots, s_n)$ returns $s_1 \oplus \ldots \oplus s_n$
- SU takes $(s_1, \ldots, s_n)$ and for $1 \leq i \leq n-1$, computes $r_i \overset{\$}{\leftarrow} \{0,1\}^\kappa$, $s_i \overset{\$}{\leftarrow} s_i \oplus r_i$. Finally, $s_n \leftarrow s_n \oplus r_1 \oplus \ldots \oplus r_{n-1}$, and SU returns $(s_1, \ldots, s_n)$.

It is immediate to see that the above scheme is correct. The following theorem states (information-theoretic) security. The proof is in the full version [11].

Theorem 1. *The scheme of Construction 41 is a refreshable secret sharing scheme with secret privacy, oblivious reconstructability and share unpredictability.*

IND-PCA SECURE ENCRYPTION. Our second construction uses a (labeled) encryption scheme that satisfies indistinguishability under plaintext-checking attacks (IND-PCA) [32]. One concrete scheme which satisfies IND-PCA security is the "Short" Cramer-Shoup (SCS) scheme proposed by Abdalla et al. [1]. We recall the primitive and the scheme in the full version [11]. The following result about IND-PCA security of the SCS scheme is by Abdalla et al. [1].

Theorem 2. *Under the DDH assumption on $\mathbb{G}$ and assuming that H is a target collision resistant hash function, the* SCS *scheme by Abdalla et al. [1] is IND-PCA.*

5 RMA Constructions

We are now ready to present two constructions of an RMA protocol for a limited, but quite practical class of attacks. The first construction combines a secret sharing scheme with a hash function, and does not rely on public key cryptography. The scheme is quite efficient and is secure in the random oracle model; the second construction uses a public key encryption scheme secure under plaintext checking attacks.

Both construction share the same underlying idea. A secret is shared and the resulting shares are placed in the memory. In our construction we assume that shares are at equal distance – other options are possible provided that this placement ensures that tampering with the memory (using the tampering functions provided to the RMA adversary) does tamper with these protective shares. The attestation protocol is challenge response: the verifier selects a random nonce and sends it to the prover. Upon receiving the nonce, the prover collects the shares, reconstructs the secret and uses it in a cryptographic operation; the verifier then confirms that the secret used is the same that he holds.

In the first scheme, which we present below, the prover hashes the secret together with the nonce and sends it to the verifier who checks consistency with his locally stored secret by and the nonce he has sent.

5.1 Hash-based RMA

Construction 51 [Hash-based RMA.] Fix a a refreshable n-out-of-n secret sharing scheme $SSh = (\mathsf{KS}, \mathsf{KR}, \mathsf{SU})$. Let Divide be any function that on input a bitstring of size greater than n breaks M into n consecutive substrings $(M_1, \ldots, M_n)$. Let $H : \{0,1\}^* \to \{0,1\}^h$ be a hash function. These scheme does not use asymmetric keys for the parties so below we omit them from the description of the algorithms. We define the RMA protocol $\mathsf{hash2rma}(H)$ by the algorithms $(\mathsf{SS}, \mathsf{Init}, (\mathsf{MA}, \mathsf{MV}), \mathsf{Update}, \mathsf{Extract})$ below:

- $\mathsf{SS}(1^k)$ returns ϵ.
- Init on input M does
  - $s \xleftarrow{\$} \{0,1\}^\kappa$
  - $(s_1, \ldots, s_n) \leftarrow \mathsf{KS}(n, s)$
  - $(M_1, \ldots, M_n) \leftarrow \mathsf{Divide}(M)$
  - Return $(M_1 \| s_1 \| \ldots \| M_n \| s_n, s)$.
- Extract on input M_s parses M_s as $M_1 \| s_1 \| \ldots \| M_n \| s_n$, runs $s \leftarrow \mathsf{KR}(s_1, \ldots, s_n)$ and returns (M, s).
- MV on input s picks $l \xleftarrow{\$} \{0,1\}^{l(\kappa)}$ and sends l to MA
- MA on input M_s gets l from MV, calculates $(M, s) \leftarrow \mathsf{Extract}(M_s)$, and sends back $t = H(s||l)$.
- MV gets t from MA returns the result of the comparison $t = H(s||l)$.
- Update on input M_s M_s as $M_1 \| s_1 \| \ldots \| M_n \| s_n$ and returns $\mathsf{SU}(s_1, \ldots, s_n)$.

The following theorem states the security guarantees the above construction provides – the details of the proof are in the full version of the paper [11].

Theorem 3. *Let $SSh = (\mathsf{KS}, \mathsf{KR}, \mathsf{SU})$ be an n-out-of-n refreshable secret sharing scheme. Let* Divide *be any function that on input a bitstring M, which for simplicity we assume is nm bits, breaks M into n consecutive substrings $(M_1, \ldots, M_n)$. Let* $\mathsf{hash2rma}(H) = (\mathsf{SS}, \mathsf{Init}, (\mathsf{MA}, \mathsf{MV}), \mathsf{Update}, \mathsf{Extract})$ *be the hash-based RMA protocol as per Construction 51.*

Let $\mathcal{L}$ be the class of functions that on inputs integers a, b such that $1 \leq a < b \leq m$, returns $M_s[a \ldots b]$. Let $\mathcal{T}$ be the class of functions that on inputs an index $1 \leq i \leq n$ and bitstring c of size $m + k$ returns M_s with its ith block changed to c.

Let us call the adversary restricted if during all its queries to Read *and* Tamper *oracles between the* Update *queries, there is a substring of M_s of length at least n, which has not been read, i.e., not returned by* Read.

Then if SSh has secret privacy, oblivious reconstructability and share unpredictability then $\mathsf{hash2rma}(H)$ *is secure wrt $\mathcal{L}$ and $\mathcal{T}$ and the adversaries restricted as above, in the random oracle model.*

We remark that while our protocol descriptions and treatment assume that the shares are embedded into the memory over equal intervals for simplicity, our implementations use blocks of increasing size, for systems functionality purposes. Our security analyses still apply though. This is because it is clear how the read and tamper queries correspond to reading and tampering the shares, and in addition, any tampering query to a memory part that has not been read must change the secret.

We justify the restrictions in the security statement from the systems point of view. We require that an attacker does not read the whole memory. This is reasonable, as reading incorrect memory address results in segmentation fault (e.g., termination of the process). Given that 64-bit address of modern processors, it's unlikely that attackers infer the whole memory space.

Since our threat model is not arbitrary memory write: rather a consecutive memory overrun like buffer overflow, it is natural to assume in this threat model an attacker needs to over-write the boundary between the blocks.

Given that the memory randomization is a common defense (outside of our model though), attackers should correctly identify the location of shares to overwrite (which is randomized), hence we do not model completely arbitrary writes.

5.2 Encryption-based RMA

The construction is based on a similar idea as that underlying the hash-based RMA protocol above. The difference is in the attestation and verification algorithms. Instead of the hash, the prover computes and sends the encryption of the secret currently encoded in the memory with the nonce sent by the verifier as label.

Construction 52 [Encryption-based RMA.] Let $SSh = (\mathsf{KS}, \mathsf{KR}, \mathsf{SU})$ and Divide be as in Construction 51. Let $\Pi = (\mathcal{K}ey\mathcal{G}en, \mathcal{E}nc, \mathcal{D}ec)$ be a labeled asymmetric encryption scheme. The RMA scheme $\mathsf{enc2rma}(\Pi)$ is defined by

- $\mathsf{SS}(1^\kappa)$ runs $(pk, sk) \overset{\$}{\leftarrow} \mathcal{K}eyGen(1^\kappa)$ and returns (pk, sk)
- Init is as in Construction 51.
- $\mathsf{Extract}$ on input M_s parses M_s as $M_1 \| s_1 \| \ldots \| M_n \| s_n$, runs $s \leftarrow \mathsf{KR}(s_1, \ldots, s_n)$ and returns (M, s).
- MV on input s picks $l \overset{\$}{\leftarrow} \{0,1\}^{l(\kappa)}$ and sends l to MA
- MA on input M_s gets l from MV and does
  - $(M, s_1, \ldots, s_n) \leftarrow \mathsf{Extract}(M_s)$,
  - $C \overset{\$}{\leftarrow} \mathcal{E}nc^l(s)$ and
  - send C to the verifier.
- MV on input C calculates $s' \leftarrow \mathcal{D}ec^l(C)$ and returns the result of $s \overset{?}{=} s'$.
- The Update algorithm is as in Construction 51.

The intuition behind security of the construction is as follows. The prover sends the encrypted secret (for some label chosen by the verifier) to the verifier; the goal of the adversary is to (eventually) create a *new* ciphertext of *the same* secret under a new label received from the verifier. If this is possible, a plaintext-checking oracle would allow to distinguish such an encryption from the encryption of a different secret. The following proposition establishes the security of the above construction. The proof is in [11].

Theorem 4. *If SSh is a refreshable secret sharing scheme with secret privacy, oblivious reconstructability and share unpredictability and $\Pi = (\mathcal{K}eyGen, \mathcal{E}nc, \mathcal{D}ec)$ is an IND-PCA secure then* $\mathsf{enc2rma}(\Pi)$ *defined by Construction 52 is a secure RMA scheme with respect to $\mathcal{L}$, $\mathcal{T}$ and any efficient but restricted adversary defined in Theorem 3.*

OPTIMIZATION. The above theorem establishes that we can instantiate an RMA scheme using the SCS scheme that we presented in Section 4. It turns out that we can further optimize the communication complexity of that protocol (where each interaction requires the prover to send three group elements) by observing that the verifier already has the plaintext that the ciphertext it receives should contain. In this case, the prover does not have to send the second component of the ciphertext (as this component can actually be recomputed by the verifier using its secret key). For completeness, we give below the relevant algorithms of the optimized scheme.

Construction 53 [SCS-based RMA.]

- $\mathsf{SS}(1^\kappa)$ obtains $\mathbb{G}$ and $(h, c, d), (x, a, b, a', b')$ by running $\mathcal{K}eyGen_{\mathsf{SCS}}(1^\kappa)$.
- MV on input s picks $l \overset{\$}{\leftarrow} \{0,1\}^{l(\kappa)}$ and sends l to MA
- MA on input M_s and (h, c, d) gets l from MV, obtains the shares of the secret via $(M, s_1, \ldots, s_n) \leftarrow \mathsf{Extract}(M_s)$, and samples random coins $r \in [|\mathbb{G}|]$ and computes $(u = g^r, e = h^r \cdot m, v = (c \cdot d^\alpha)^r)$, where $\alpha = H(l, u, e)$. It sends (u, v) to the server.
- MV on input its secret key (x, a, b, a', b') the challenge l and secret s operates as follows on input (u, v) from the prover and returns the result of the comparison $v = u^{a+\alpha a'} \cdot (u^x)^{b+\alpha b'}$, where $\alpha = H(l, u, u^x \cdot s)$.

The following security statement follows directly from Theorem 4 and Theorem 2.

Theorem 5. *If SSh is a refreshable secret sharing scheme with secret privacy, oblivious reconstructability and share unpredictability, and $\Pi = (\mathcal{K}eyGen, \mathcal{E}nc, \mathcal{D}ec)$ is as per Construction 52 then the RMA protocol defined by Construction 53 is a secure RMA scheme with respect to $\mathcal{L}$, $\mathcal{T}$ and any efficient but restricted adversary defined in Theorem 3, assuming the DDH problem is hard in the underlying group and the hash is target collision-resistant.*

6 Implementation and Evaluation

Our prototype can seamlessly enable the remote memory attestation in any applications that are using standard libraries. At runtime, the prototype implementation interposes all memory allocations (`malloc()`) and deallocations (`free()`) by incorporating `LD_PRELOAD` when the application starts executing. Before the application runs, our custom runtime pre-allocates memory regions with varying sizes, and carefully insert key shares between the memory objects.

Specifically, we provide a simple wrapper program (called *prover*) which end users use to perform all these operations. When requested, the prover launches the program, and then inserts our custom library for memory allocations of the target application. Before the program starts, the prover pre-allocates a list of chuncked memory, starting from 8 bytes object to a few mega bytes (128 MB by default) incrementally. In our current prototype, we pre-allocate N blocks (configurable, 10 by default) per size (e.g., N 8-byte blocks up to 128 MB).

For attestation, the prover initiates the secrets with the public key provided, performs the memory attestation of the program it launched, and communicates with the remote verifier. To access the memory of a remote program, it attaches to the program via `ptrace` interface in UNIX-like operating system, and runs the protocol.

Component	**Lines of code**	
Verifier	298	lines of C
Prover	638	lines of C
Memory allocator	343	lines of C
Total	1,279	lines of code

Fig. 4: The complexity of RMA in terms of lines of code of each components, including verifier, launcher and memory allocator.

We evaluate a prototype of RMA in three aspects: 1) runtime overheads of computation-oriented tasks such as SPEC benchmark; 2) worst case overheads (e.g., launching an application) that end-user might be facing when using RMA; 3) break-down of performance overheads and data transferred on the course of remote attestation by using our prototype. We performed all experiments with the prototype implementation of the

Programs	Baseline (s)	RMA (s)	Overhead (%)
400.perlbench	545	566	3.9%
401.bzip2	749	770	2.8%
403.gcc	521	537	3.1%
429.mcf	385	395	2.6%
445.gobmk	691	691	0.0%
456.hmmer	638	665	4.2%
458.sjeng	779	805	3.3%
462.libquantu	1,453	1,514	4.2%
464.h264ref	917	950	3.6%
471.omnetpp	540	547	1.3%
473.astar	606	635	4.8%
483.xalancbmk	361	373	3.3%

Fig. 5: Runtime overheads of SPEC benchmark programs with RMA.

encryption-based RMA. As we mentioned in the Introduction, this protocol is not as efficient (in terms of crypto operations) as the hash-based one, but it provides stronger security (no reliance on the random oracle model), and performs equally well in the presence of system-dependent overheads.

Micro-benchmark. We evaluate a prototype of RMA by running the standard SPEC CPU2006 integer benchmark suite. All benchmarks were run on Intel Xeon CPU E7-4820 @2.00GHz machine with 128 GB RAM, and the baseline benchmark ran with standard libraries provided by Ubuntu 15.04 with Linux 3.19.0-16. As shown in Figure 5, due to the simplicity of the implementation, RMA incurs negligible performance overheads to SPEC benchmark programs: 3.1% on average, ranging from 0.0% to 4.8% depending on a SEPC benchmark program. During the experiments, we found out that the significant part of performance overheads comes from the implementation of the custom memory allocator and the side-effects of memory fragmentation, thereby diluting the overheads related to crypto operations. We believe that different types of applications requiring frequent validation or updates of share keys might need better optimization of crypto-related software stack. It is worth noting that our prototype never focuses on optimization in any sort (e.g., using a coarse-grained, global lock to support multi-threading) and the overall performance can be dramatically improved if necessary.

Macro-benchmark. To measure performance overhands that end-user might be encountering when using RMA, we construct a macro-benchmark with three applications for four different tasks; launching a web browser (Firefox), an email client (Thunderbird), compressing and decompressing files (Tar). All experiments were conducted on a laptop running Ubuntu 12.04 with standard `glibc` library (Ubuntu/Linaro 4.6.3-1ubuntu5), and we measured each benchmark ten times, we provide the summary in [11]. iNote that launching application is the worst-case scenario to RMA because it has to allocate memory space at program's startup and initiate all key shares before executing the program. According to our benchmark, it incurs acceptable performance overheads even in the worst-cast construction, but we believe the latency that users actually feel is minimal: 0.023s in Firefox and 0.199s in Thunderbird.

PERFORMANCE BREAK-DOWN. We also measured how long it takes to proceed each stage of the RMA protocol with our prototype implementation. We measured the amount of data that needs to be transferred as well. In short, it is feasible to implement the proposed RMA protocol in practice: our unoptimized system incurs negligible performance overheads (the details are in [11]) and the amount of messages between the prover and the verifier is minimal (e.g., 12 bytes up to 396 bytes). According to our evaluation, we believe our RMA protocol can be utilized in an efficient manner in practice.

7 Acknowledgements

The first author was supported in part by the NSF award CNS-1422794. The fourth author was supported in part by European Union Seventh Framework Programme (FP7/2007-2013) grant agreement 609611 (PRACTICE). We thank Sangmin Lee for great help with implementations. We thank Tom Conte and Milos Prvulovic for useful discussions and Rafail Ostrovsky and Vassilis Zikas for clarifications on [26].

Bibliography

[1] M. Abdalla, F. Benhamouda, and D. Pointcheval. Public-key encryption indistinguishable under plaintext-checkable attacks. In J. Katz, editor, *Public-Key Cryptography - PKC 2015 - 18th IACR International Conference on Practice and Theory in Public-Key Cryptography Proceedings*, volume 9020 of *Lecture Notes in Computer Science*, pages 332–352. Springer, 2015.

[2] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[3] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann. A security framework for the analysis and design of software attestation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1–12. ACM, 2013.

[4] R. Barnett. GHOST gethostbyname() heap overflow in glibc (CVE-2015-0235), Jan. 2015. `https://www.trustwave.com/Resources/SpiderLabs-Blog/GHOST-gethostbyname()-heap-overflow-in-glibc-(CVE-2015-0235).`

[5] M. Bellare, D. Cash, and R. Miller. Cryptography secure against related-key attacks and tampering. In D. H. Lee and X. Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 486–503. Springer, 2011.

[6] M. Bellare and T. Kohno. A theoretical treatment of related-key attacks: RKA-PRPs, RKA-PRFs, and applications. In *Advances in Cryptology EUROCRYPT 2003 Proceedings*, pages 491–506. Springer, 2003.

[7] M. Bellare, K. G. Paterson, and S. Thomson. RKA security beyond the linear barrier: IBE, encryption and signatures. In X. Wang and K. Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 331–348. Springer, 2012.

[8] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM, 1993.

[9] E. D. Berger. HeapShield: Library-based heap overflow protection for free, 2006. University of Massachusetts Amherst, TR 06-28.

[10] R. Bhattacharyya and A. Roy. Secure message authentication against related-key attack. In S. Moriai, editor, *Fast Software Encryption - 20th International Workshop, FSE 2013. Revised Selected Papers*, volume 8424 of *Lecture Notes in Computer Science*, pages 305–324. Springer, 2013.

[11] A. Boldyreva, T. Kim, R. Lipton, and B. Warinschi. Provably-secure remote memory attestation to prevent heap overflow attacks. Full version of this paper. Cryptology ePrint Archive, Report 2015/729, 2015. `http://eprint.iacr.org/2015/729`.

[12] R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited. *Journal of the ACM (JACM)*, 51(4):557–594, 2004.

[13] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, 2009.

[14] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, 1998.

[15] R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *Advances in Cryptology – CRYPTO'98*, pages 13–25. Springer, 1998.

[16] L. Duflot, Y.-A. Perez, and B. Morin. What if you can't trust your network card? In *Recent Advances in Intrusion Detection*, pages 378–397. Springer, 2011.

[17] H. Etoh. GCC extension for protecting applications from stack-smashing attacks (ProPolice), 2003. `http://www.trl.ibm.com/projects/security/ssp/`.

[18] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A minimalist approach to remote attestation. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014*, pages 1–6, 2014.

[19] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proc. of the 10th Usenix Security Symposium*, pages 55–66, 2001.

[20] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In D. Coppersmith, editor, *Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference Proceedings*, volume 963 of *Lecture Notes in Computer Science*, pages 339–352. Springer, 1995.

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

[22] M. Jakobsson and K.-A. Johansson. Practical and secure software-based attestation. In *Lightweight Security & Privacy: Devices, Protocols and Applications (LightSec), 2011 Workshop on*, pages 1–9. IEEE, 2011.

[23] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for timing-based attestation. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 239–253. IEEE, 2012.

[24] Y. Li, J. M. McCune, and A. Perrig. SBAP: Software-based attestation for peripherals. In *Trust and Trustworthy Computing*, pages 16–29. Springer, 2010.

[25] Y. Li, J. M. McCune, and A. Perrig. Viper: verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 3–16. ACM, 2011.

[26] R. J. Lipton, R. Ostrovsky, and V. Zikas. Provable virus detection: Using the uncertainty principle to protect against malware. Cryptology ePrint Archive, Report 2015/728, 2015. `http://eprint.iacr.org/`.

[27] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. ASLR-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015.

[28] 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 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, 2010.

[29] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, 2008.

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

[31] N. Nikiforakis, F. Piessens, and W. Joosen. Heapsentry: Kernel-assisted protection against heap overflows. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 10th International Conference, DIMVA 2013 Proceedings*, pages 177–196, 2013.

[32] T. Okamoto and D. Pointcheval. REACT: rapid enhanced-security asymmetric cryptosystem transform. In D. Naccache, editor, *Topics in Cryptology - CT-RSA 2001, The Cryptographer's Track at RSA Conference 2001 Proceedings*, volume 2020 of *Lecture Notes in Computer Science*, pages 159–175. Springer, 2001.

[33] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *Proceedings of the 17th USENIX Conference on System Administration*, LISA '03, 2003.

[34] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, 2012.

[35] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, 2005.

[36] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 272–282. IEEE, 2004.

[37] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, 2004.

[38] H. Wee. Public key encryption against related key attacks. In *Public Key Cryptography - PKC 2012 - 15th International Conference on Practice and Theory in Public Key Cryptography Proceedings*, pages 262–279, 2012.

[39] Y. Younan, W. Joosen, and F. Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *ICICS*, volume 4307 of *Lecture Notes in Computer Science*, pages 379–398. Springer, 2006.