Provably-Secure Remote Memory Attestation for Heap Overflow Protection

ALEXANDRA BOLDYREVA[*] TAESOO KIM[†] RICHARD LIPTON[‡]
BOGDAN WARINSCHI[§]

Abstract

We initiate the study of *provably secure* remote memory attestation to mitigate heap-based overflow attacks. We present two protocols offering various efficiency and security trade-offs 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, and should be useful for further research on the subject of matter.

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 heap overflow and use-after-free vulnerabilities. Without question, this problem is of great importance and has been extensively studied by the security community. Existing solutions (e.g., stack and heap canaries [16, 19, 20], address space layout randomization (ASLR) [38], 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 exiting 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 non-negligible 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. Accordingly, without a clear adversarial model it is hard to judge the scope of the protection,

[*] Georgia Institute of Technology. E-mail: `sasha@gatech.edu`.
[†] Georgia Institute of Technology. E-mail: `taesoo@gatech.edu`.
[‡] Georgia Institute of Technology. E-mail: `rjl@cc.gatech.edu`.
[§] University of Bristol. E-mail: `csxbw@bristol.ac.uk`.

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 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 satisfying realistic threats and 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.

1.1 Our Focus

Our focus is on the *remote* verification setting, motivated by the widespread 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 [6]), and become more important in the cloud setting where we have to rely on the cloud provider.

The prevalent solutions that insert "canaries" into memory and verify their integrity later [5, 16, 19, 20, 32], 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.

As we explain later in the paper, our solutions could be viewed as a novel variant of cryptographic canaries, suitable for remote setting and providing provable security guarantees under precisely defined threat models.

1.2 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 no security may be possible if an attacker's queries are unrestricted. 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 [6] 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.

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.

1.3 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. Let's assume for simplicity for now 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.

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 an ideal setting, we would refresh the shares whenever the leakage of the secret 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 it as a parameter (e.g., certain time period) and developers can simply incorporate proper timing with our implementation.

Despite the solution approach above being so simple and sound, it turns out that assessing its security and practicality needs to deal with 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 [21]. 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 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. However, the proof we provide relies on the idealistic random oracle (RO) model [10]. It is known that in principle, protocols proven secure in the RO model may have no secure "real" hash instantiation [14]. Therefore 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 even 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) [8]. 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. [7, 13], only address specific algebraic classes of tampering functions and are rather inefficient.

Somewhat unexpectedly, we utilize a public key encryption scheme for encrypting the random challenge and 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 [9, 40]).

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 memory protection 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 [17] 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 [33] 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 show how one can optimize further and save an additional group element in the communication by slightly increasing computation.

1.4 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 negligible 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.

1.5 Related Work

Canary-based protection has been adopted to prevent stack smashing [2]: e.g., ProPolice [19], StackGuard [16], StackGhost [20]. Similarly, canaries (or guard as a general form) have been used for heap protection, in particular metadata of heap [34, 41] (e.g., double free): HeapSheild [12] or AddressSanitizer [35]. Unlike these practical measures, the main goal of RMA is to provide a provable guarantee of the memory integrity, under the context of software-based remote attestation.

Software-based attestation has been explored in various contexts: peripheral firmware [18, 25, 27], embedded devices [15, 26, 37], or legacy software [36]. 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 adveresarial inteference. 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, counstructions employ carefully crafted check-sum functions where unforgeability *heuristically* relies on timing assumptions and lack of storage space on the device. Jacobsson and Johansson [24] show that such assumptions can be grounded in the assumptions that RAM access is faster than access to the secondary storage [24]. Our work is similar in its goals with that of Armknecth et al. [4] 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 [30] and TrustVisor [29] using TPM, InkTag [23] based on a hypervisor, and Haven using Intel SGX [3, 22, 31]. Our work differs in that we do not explicitly rely on hardware assumptions and is built on top of the provable security guarantee.

Finally, a recent paper [28] 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 [28] are quite different from ours. The major difference is that the attacker in the security model of [28] 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 [28] is mostly of theoretical interest, while our solution is quite efficient. The work [28] has additional results about protection against tiny overwrites but that requires CPU modifications.

1.6 Outline

We start with explaining the notation in Section 2. Next we define the functionality and security of Remote Memory Attestation (RMA) Protocols in Section 3. Section 4 presents the building blocks for our constructions. In Section 5 we present two RMA protocols and prove that they satisfy our security model. The first protocol is quite efficient, and its security is based on the random oracle model. Security of our second construction relies only on standard computational assumptions, and is less efficient (but still practical). Finally, in Section 6 we present our implementations results and follow with conclusions in Section 8.

2 Notation

We denote by $\{0,1\}^*$ the set of all binary strings of finite length. If X is a string, then $|X|$ denotes its length in bits. If S is a set, then $|S|$ denotes the size of S; $X \overset{\$}{\leftarrow} S$ denotes that X is selected uniformly at random from S. If A is a randomized algorithm, then the notation $X \overset{\$}{\leftarrow} A$ 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

3.1 Syntax

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

Definition 3.1. [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* $(\mathsf{MA}, \mathsf{MV})$*, ran 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.

3.2 RMA Security

We now formally define the security model for an RMA protocol, which is part of our main contributions. This step is essential for designing schemes that provide security guarantees.

We consider an attacker who can read the public key (if any), and can observe the interactions between the prover and the verifier. But our adversary is significantly more powerful. In the first stage of its attack, the attacker can read arbitrary parts of the memory and can over-write a part of the memory by injecting a data (code) of its choosing. Importantly, the adversary can intercept and modify the communication between the prover and the verifier. This is captured by giving the adversary oracle access to the oracles that follow the interactive RMA protocol, while the adversary can chose to observe a legitimate protocol by forwarding the answers of one oracle to another; or it can choose to manipulate the conversation, or even supply inputs of its own choosing. Also, the attacker can request to do an update at any point. 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. 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, e.g., if the adversary reads the whole memory in between the secret updates or reads a block and immediately over-writes it so that the secret share is intact. Moreover, note that the adversary which 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 observing the corresponding response by the prover and the verifier's decision.

Second, it seems difficult to have a single solution that protects against a very wide class of attacks. Hence, it makes sense to have multiple solutions for various applications and classes of attacks. But instead of having multiple security definitions tailored for each case, one can have a rather general easily "customizable" definition.

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

We now present the formal definition and then follow with further informal explanations of how the definition captures practical threats.

Definition 3.2. [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

```
$\mathbf{Exp}_{A,\Pi}^{\text{rma-}(\mathcal{L},\mathcal{T})}$:
    $(pk, sk) \leftarrow \mathsf{SS}$
    $M \leftarrow A(pk)$
    $(M_s, s) \leftarrow \mathsf{Init}(M, pk, sk)$
    $g' \leftarrow A^{\mathsf{Read}(\cdot),\mathsf{Tamper}(\cdot),\mathsf{MA}(pk,M_s),\mathsf{MV}(sk,s),\mathsf{Update}}$
    $M_s \leftarrow \mathsf{Update}(M_s)$
    $M_s \leftarrow g'(M_s)$
    $A^{\mathsf{Tamper}(\cdot),\mathsf{MA}(pk,M_s),\mathsf{MV}(sk,s)}$
    Output 1 iff 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 $\perp$
    $M_s \leftarrow g(M_s)$
```

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

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

Both our constructions use as a building block a secret sharing scheme, which we now define.

4.1 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 [21]. In addition, we will require two extra property we discuss later.

4.1.1 Syntax

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

Definition 4.1. *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*[1]*. 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*[2]*. 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.*

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

[2]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 $\mathsf{SU}(\mathsf{SU}(\ldots \mathsf{SU}((s_1, s_2, \ldots, s_n)) \ldots))$.

4.1.2 Security

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

SECRET PRIVACY. The most basic one, privacy for the secret, is defined via the experiments associated with adversary A and parameterized with a bit $b \in \{0,1\}$, presented in Figure 2. Our definition uses a different style than that of [21], as the latter targets sharing of algebraic keys for public key operations.

Here we consider an adversary who can adaptively learn at most $n-1$ key shares; at this point the shares are refreshed using the KR algorithm and the adversary has the possibility to learn more shares, and so forth.

```
$\mathbf{Exp}^{\text{rssh-}b}_{A,\Pi}$:
  $j \leftarrow 0$
  $s, s' \xleftarrow{\$} A$
  If $s, s' \notin \{0,1\}^\kappa$ then abort
  If $b = 0$ then $(s_1, s_2, \ldots, s_n) \xleftarrow{\$} \mathsf{KS}(s)$,
      otherwise $(s_1, s_2, \ldots, s_n) \xleftarrow{\$} \mathsf{KS}(s')$.
  $d \xleftarrow{\$} A^{\mathsf{SomeShares}_b(\cdot)}$
  Return d

Oracle $\mathsf{SomeShares}_b(i)$:
  if $i < 1$ or $i > n$ then return $\perp$
  $j \leftarrow j + 1$
  If $j = n$ then
      $(s_1, s_2, \ldots, s_n) \xleftarrow{\$} \mathsf{SU}((s_1, s_2, \ldots, s_n))$
      $j \leftarrow 0$
  Return $s_i$
```

Figure 2: Games defining the security of refreshable secret sharing scheme $\Pi = (\mathsf{KS}, \mathsf{KR}, \mathsf{SU})$

We say that $(\mathsf{KS}, \mathsf{KR}, \mathsf{SU})$ is refreshable secret sharing scheme with secret privacy if for every adversary A the distributions of its output are statistically close in both experiments.

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.

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 3, 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.

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 predict the value of any of the resulting fresh shares. This intuition is formalized using the game $\mathbf{Exp}^{\text{unpred}}_{A,\Pi}$ in Figure 4. We say that Π satisfies share unpredictability if for any adversary the probability that the experiment returns 1 is nonnegligible.

4.1.3 Secure Construction

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

```
$\mathbf{Exp}^{\mathrm{rec}}_{A,\Pi}$:
    $s \xleftarrow{\$} A; L \leftarrow [\,]$
    $\{s_1, s_2, \ldots, s_n\} \leftarrow \mathsf{KS}(s)$
    $A^{\mathsf{ShareInfo}(\cdot)}$
    $b \leftarrow SC(L)$
    $s' \leftarrow \mathsf{KR}(s_1, s_2, \ldots, s_n)$
    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 $(i, v) \in \{1, 2, \ldots, n\} \times \{0,1\}^\kappa$
    $L \leftarrow L :: (i, v)$
    $s_i \leftarrow v$
```

Figure 3: Game defining the oblivious reconstruction property for secret sharing scheme $\Pi = (\mathsf{KS}, \mathsf{KR}, \mathsf{SU})$

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

    Oracle $\mathsf{Read}(i)$
      return $s_i$

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

Figure 4: Game defining share unpredictability for for secret sharing scheme $\Pi = (\mathsf{KS}, \mathsf{KR}, \mathsf{SU})$. We demand that A quries his Tamper at least once.

Construction 4.2. [Refreshable secret sharing] *We define the scheme* $(\mathsf{KS}, \mathsf{KR}, \mathsf{SU})$ *as follows.*

- KS *takes secret* $s \in \{0,1\}^\kappa$, *picks* $s_i \xleftarrow{\$} \{0,1\}^\kappa$ *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* $r_i \xleftarrow{\$} \{0,1\}^\kappa$, $s_i \xleftarrow{\$} s_i \oplus r_i$. *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.

Theorem 4.3. *The scheme of Construction 4.2 is a refreshable secret sharing scheme with secret privacy, oblivious reconstructability and share unpredictability.*

Proof. The proof for the first part and third requirements is trivial and is omitted. To prove the oblivious reconstruction property, we consider an algorithm SC which works as follows. Given the list L of queries that the adversary A makes, it checks for the following invariant. If the adversary has replaced a share without having read it first then SC returns 0 (i.e. the secret has changed). Otherwise, let I be the set of indexes for those shares that the adversary has replaced (and which therefore has also read). Let p be $\oplus_{i \in I} s_i$ where s_i is the original value for share i; let v_i be the current value for share i and let $v = \oplus_{i \in I} v_i$. The reconstruction algorithm returns 1 if $p = v$ and 0 otherwise. The intuition behind the construction of SC is that if the adversary overwrites a share without knowing its value then most likely the final value of the secret will be different from the original one. The same holds true if the adversary changes the value encoded by the shares in I. □

4.2 IND-PCA Secure Encryption

Our second construction uses a (labeled) encryption scheme that satisfies indistinguishability under plaintext-checking attacks (is IND-PCA) [33]. We recall the primitive and the security definition.

LABELED ENCRYPTION. A labeled encryption scheme is given by algorithms $(\mathcal{K}eyGen, \mathcal{E}nc, \mathcal{D}ec)$ where $\mathcal{K}eyGen$ is like in a standard asymmetric encryption scheme but the remaining algorithms take an additional input, a *label*. We write $\mathcal{E}nc^l(pk, x)$ for encrypting plaintext x with respect to label l under public key pk and write $\mathcal{D}ec^l(sk, c)$ for decrypting ciphertext c with respect to label l using secret key sk. Correctness of such schemes demands that for any pk, sk output by the key generation algorithm, and label l and any plaintext x it holds that $\mathcal{D}ec^l(sk, \mathcal{E}nc^l(pk, x)) = x$.

INDISTINGUISHABILITY UNDER PLAINTEXT-CHECKING ATTACKS. Roughly, such schemes guarantee security against adversaries who can test if a ciphertext encodes a certain plaintext. This notion is formalized below.

Definition 4.4. *Let* $\Pi = (\mathcal{K}eyGen, \mathcal{E}nc, \mathcal{D}ec)$ *be an asymmetric encryption scheme and consider the experiment in Figure 5. We say that* Π *satisfies indistinguishability of ciphertexts under plaintext-checking attacks (*ind-pca*) if for any adversary efficient adversary* A *its advantage* $\mathbf{Adv}^{\mathsf{OPCA}}_{A,\Pi}(1^\kappa)$ *defined by:*

$$\Pr\left[\mathbf{Exp}^{\text{ind-pca-1}}_{A,\Pi}(1^\kappa) = 1\right] - \Pr\left[\mathbf{Exp}^{\text{ind-pca-0}}_{A,\Pi}(1^\kappa) = 1\right]$$

is negligible.

```
$\mathbf{Exp}^{\text{ind-pca-}b}_{A,\Pi}$:
  $(pk, sk) \leftarrow \mathcal{K}eyGen$
  $(l^*, x_0, x_1) \leftarrow A(pk)$
  $C^* \leftarrow \mathcal{E}nc^{l^*}(pk, x_b)$
  $b' \leftarrow A^{\mathsf{OPCA}(\cdot,\cdot)}(C^*)$
  Return $b'$

Oracle $\mathsf{OPCA}(x, (l, c)))$:
  $x' \leftarrow \mathcal{D}ec^{l^*}(sk, c)$
  if $x = x'$ return 1
  else return 0
```

Figure 5: Game defining indistinguishability under plaintext-checking attacks for the security of labeled encryption scheme $\Pi = (\mathcal{K}eyGen, \mathcal{E}nc, \mathcal{D}ec)$.

IND-PCA SCHEME. One concrete scheme which satisfies ind-pca security is the "Short" Cramer-Shoup (SCS) scheme proposed and analyzed by Abdalla et al. [1] which we recall bellow.

Construction 4.5 (Short Cramer-Shoup (SCS))**.** *The algorithms of the scheme use a collision resistant hash function H and are as follows.*

- *The key-generation, on security parameter κ, outputs a group $\mathbb{G}$ together with generators g, h (for security parameter κ). It selects a secret key $sk = (x, a, b, a', b')$ at random from $[|\mathbb{G}|]$. The corresponding public key is $(c, d, h) = (h = g^x, c = g^a h^b, d = g^{a'} h^{b'})$.*

- *The encryption of m with label l under public key (c, d, h) is obtained by sampling random coins $r \in [|\mathbb{G}|]$ and computing $C = (u = g^r, e = h^r \cdot m, v = (c \cdot d^\alpha)^r)$, where $\alpha = H(l, u, e)$.*

- *To decrypt using secret key $sk = (x, a, b, a', b')$ the ciphertext $C = (u, e, v)$ for label l one computes $m \leftarrow e/u^x$ and checks that $v = u^{a+\alpha a'}(e/m)^{b+\alpha b'}$, where $\alpha = H(l, u, e)$. If the equality succeeds the decryption outputs m, otherwise it outputs $\perp$.*

The following result about ind-pca security of the SCS scheme is by Abdalla et al. [1].

Theorem 4.6. *Under the DDH assumption on $\mathbb{G}$ and assuming that H is a target collision resistant hash function, the above scheme is indistinguishable under plaitext checking attacks.*

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 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 5.1. [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 $H : \{0,1\}^* \rightarrow \{0,1\}^h$ 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* $(M_1\|s_1\|\ldots\|M_n\|s_n, s)$.

- $\mathsf{Extract}$ *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* $l \xleftarrow{\$} \{0,1\}^{l(\kappa)}$ *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.

Theorem 5.2. *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 5.1.*

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* i*th 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.

Proof. Our proofs follow the "game hopping" technique [11, 39] by considering a sequence of games associated with an adversary.

In the description of the games below we write s^0 for the secret that is selected in the initialization phase (i.e. the secret whose shares are placed in the memory). The proof relies on the observation that for the class of RMA adversaries that we consider, we can translate any query of an adversary towards its Read oracle to a query reading a share (of the secret) from the memory; similarly we translate each query of the adversary to its Tamper oracle to a query overwriting a share of the secret with some constant that the adversary chooses. Key to our proof is that the restriction we place on a valid RMA adversary implies that at any point during the execution there is at least one share which is not red by the adversary and which is in fact unpredictable. In turn, this implies that the secret itself is unpredictable and therefore the adversary should not be able to convince the verifier that it “knows" it. The difficulty in the proof is to argue the unpredictability of the secret as the adversary can intertwine reading, writing, and updating the shares.

The games that we use are as follows.

- **Game 0.** This game is the same as $\mathbf{Exp}^{\text{rma-}(\mathcal{L},\mathcal{T})}_{A,\mathsf{hash2rma}(H)}$.
- **Game 1.** In this game we only add some bookkeeping information useful for later simulations. We maintain the list O of operations (read and write actions) to memory shares that correspond to the queries that the attacker makes to its Read, Tamper oracles (per the assumption stated above).

  Notice that if the underlying secret sharing scheme has the oblivious reconstruction property then applying the algorithm SC (ensured by the oblivious reconstruction property) to the list O indicates, at any point, if the queries of the adversary have modified the secret encoded by the shares in the memory. In addition, by applying SC to any contiguous sublist of $O[i, \ldots, j]$ indicates if the secret encoded by the shares right before entry i is added to O is the same or not as the secret encoded by the shares, immediately following the addition of j'th entry to O.

  Since in this game we only add the additional bookkeeping without affecting the output, the outputs of Game 0 and Game 1 are indistinguishable.
- **Game 2.** In this game, we keep track (by using the list O) of the queries with which the adversary interacts with the shares in the memory. If at any point the adversary tampers with a share that it did not read we set the MV oracle to reject all queries in the second phase.

  We claim that an adversary that wins in Game 1 also wins in Game 2: an adversary would observe a difference between the games if it tampers with some share without reading it and yet it manages to make oracle MV accept (in the second phase) in Game 1 – by construction the oracle would reject in Game 2. Since the second phase of the game immediately follows an update of the shares, the only way for the adversary to win is tamper with the memory (which guarantees that he tampers with at least a share that did not read). This breaks the share unpredictability property of the underlying secret sharing scheme.
- **Game 3.** This game is like Game 2, except that we modify the behavior of oracle MA when queried at a moment when the secret encoded in the shares is different from the original one. More precisely, whenever this is the case, the MA oracle will use a “fake" secret f^i selected independently at random from $\{0,1\}^\kappa$; we could simply select such a secret as soon as we detect that the underlying secret has changed. However, the simulation needs to take care of the situation where MA is queried twice with the same value at moments when the underlying secret is the same (but different from the original value). We proceed as follows.

  We will associate to each entry in the list O (which corresponds to a change in the underlying secret) a new secret as follows. Whenever the j'th entry is added to O (the adversary may attempt to tamper with the secret encoded in the shares) we check weather $SC(O[1, \ldots, j]) = 1$; if this is the case we set $f^j = s^0$ (which indicates that at this point the underlying secret had not changed). Otherwise we determine if $O([i, \ldots, j]) = 1$ for some $1 < i < j$; if this is the case we set $f^j = f^i$, otherwise we select a fresh secret $f^j \xleftarrow{\$} \{0,1\}^\kappa$. The game maintains all of these values; at any point, we let f^* be the fake secret associated to the latest entry in O. It is useful for the discussion below to also introduce notation for the value of the secret that is encoded by the shares at the different points in execution. We therefore write r^j for (the real) value that is encoded by the shares when entry j is made on list O and, similarly write r^* for its value at the current point in the execution.

The execution of the game uses these secrets as follows. Whenever the adversary issues query l to oracle MA we check if the secret has not changed (i.e. if $SC(O) = 1$). If this is the case MA works normally (i.e. uses s^0); otherwise instead of reconstructing the current value r^* of the underlying secret we let MA use s^*. The rest of the game is unchanged.

We claim that any RMA adversary wins Game 2 about as often as it wins Game 3. The intuition is that the difference between the games consists in how the oracles MA answer their queries: in Game 1 the oracles use the value r^* whereas in Game 2 the oracles use the value f^*. The only way for the adversary to notice the difference in the simulation is to query the random oracle on some value $(r^i||l)$ – however, an adversary that issues such a query breaks the secrecy/share unpredictability of the underlying shares (as at lest one of the secret shares is hidden from its view).

- **Game 4.** Finally in this game, we modify oracles MA and MV to use a freshly selected secret s^1 (secret s^0 is still used in the initialization phase). Technically, we modify the game above as follows: we select a fresh secret s^1 which we associate to all positions i in O for which $SC(O[1, \ldots, i]) = 1$. Whenever the adversary queries MA we let the answer be $H(s^*||l)$. Oracle MV always uses s_1 for it checks.

  Notice that the difference between Game 3 and Game 4 is the relation between the value that MA and MV use as initial secret and the shares that are placed in the memory: in the first game the shares and the secret are related whereas in the second game they are not. We show that if an adversary distinguishes between the two games then the adversary breaks secrecy property of the secret sharing scheme.

  We argue that in Game 4 the adversary has negligible advantage to win. The restrictions on the adversary demands that the tamper function that it sends before the last stage changes one unpredictable share. In turn, this implies that in the second phase MA will use a secret independent from s^1 (which is the secret held by the verifier). Since the verifier produces (with overwhelming) probability a nonce that is distinct from those used in phase 1 and the view of the advesary is independent of s^1 (unless it queries $H(s^1||l)$ to its oracle) we conclude that the RMA adversary does not win in Game 4, except with negligible probability.

  To conclude the proof, we claim that by the game-hopping technique, the adversary can win in the original game only with negligible probability, under the assumptions stated in the theorem. And hence the theorem statement follows.

$\square$

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 5.3. [Encryption-based RMA.] *Let $SSh = (\mathsf{KS}, \mathsf{KR}, \mathsf{SU})$ and* Divide *be as in Construction 5.1. 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 $(pk, sk) \xleftarrow{\$} \mathcal{K}ey\mathcal{G}en(1^\kappa)$ and returns (pk, sk)*
- Init *is as in Construction 5.1.*
- 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 *and does*
  - $(M, s_1, \ldots, s_n) \leftarrow \mathsf{Extract}(M_s)$,
  - $C \xleftarrow{\$} \mathcal{E}nc^l(s)$ *and*
  - *send C to the verifier.*
- MV *on input C calculates $s' \leftarrow \mathcal{D}ec^l(C)$ and returns the result of the comparison $s \overset{?}{=} s'$.*
- *The* Update *algorithm is as in Construction 5.1.*

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.

Theorem 5.4. *If SSh is a refreshable secret sharing scheme with secret privacy, oblivious reconstructability and share unpredictability and $\Pi = (\mathcal{K}ey\mathcal{G}en, \mathcal{E}nc, \mathcal{D}ec)$ is an* ind-pca *secure then* $\mathsf{enc2rma}(\Pi)$ *defined by Construction 5.3 is a secure RMA scheme with respect to $\mathcal{L}$, $\mathcal{T}$ and any efficient but restricted adversary defined in Theorem 5.2.*

The proof of the theorem shares many of the steps we the proof of Theorem 5.2. In particular, Games 1-4 are obtained in a similar manner. We need to adapt the argument that concerns the gap between Game 3 and Game 4. For the encryption-based scheme we note that, informally, the difference between the two games is that the encryption and verification algorithms use secrets s^0 (in Game 3) and s^1 (in Game 4); the shares used in the initialization procedure are, in both games, shares of s^0. Therefore, an adversary that can differentiate between the two games can actually distinguish between ciphertexts of s^0 and ciphertexts of s^1 We turn this intuition into an adversary who can break ind-pca security of the underlying encryption scheme. The reduction selects two secrets s^0 and s^1 and simulates the RMA scheme to an adversary (as in Game 4). In particular we use the left-right encryption oracle to simulate the behaviour of the MA oracle, and use the plaintext checking oracle to simulate the behavior of MV. The distance between Games 3 and 4 is therefore the advantage of an adversary against ind-pca security.

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 5.5. [SCS-based RMA.]

- $\mathsf{SS}(1^\kappa)$ *obtains* $\mathbb{G}$ *and* $(h, c, d), (x, a, b, a', b')$ *by running* $\mathcal{K}ey\mathcal{G}en_{\mathsf{SCS}}(1^\kappa)$.
- 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 5.6 and Theorem 4.6.

Theorem 5.6. *If* SSh *is a refreshable secret sharing scheme with secret privacy, oblivious reconstructability and share unpredictability. and* $\Pi = (\mathcal{K}ey\mathcal{G}en, \mathcal{E}nc, \mathcal{D}ec)$ *is as per Construction 4.5 then the RMA protocol defined by Construction 5.5 is a secure RMA scheme with respect to* $\mathcal{L}$, $\mathcal{T}$ *and any efficient but restricted adversary defined in Theorem 5.2, assuming the DDH problem is hard in the underlying group and the hash is target collision-resistant.*

6 Implementation

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.

In specific, end users needs to perform all these procedures by using a simple wrapper program, called *prover*, that we provide. 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 attention 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

Figure 6: The complexity of RMA in terms of lines of code of each components, including verifier, launcher and memory allocator.

7 Evaluation

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 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 presense of system-dependent overheads.

7.1 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 Table 1, RMA incurs negligible performance overheads: 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.

7.2 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 experiment is conducted in a laptop running Ubuntu 12.04 with standard `glibc` library (Ubuntu/Linaro 4.6.3-1ubuntu5), and we measured each benchmark ten times (see Table 2). Note 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.

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%

Table 1: Runtime overheads of SPEC benchmark programs with RMA. While our memory allocator causes memory fragmentation, the simplicity of the implementation incurs negligible performance overheads to SPEC benchmark programs, from 0.0% in `gobmk` up to 4.8% in `astar`.

Programs	Baseline (s)	RMA (s)	Overhead (%)	Description
Firefox	1.4283	1.4511	1.6%	Launch Firefox with an empty page
Thunderbird	1.2467	1.4455	15.9%	Launch Thunderbird
Tar (compress)	0.7857	0.7635	-2.8%	Compress 74 MB of data with Tar
Tar (decompress)	0.7397	0.8169	10.4%	Decompress 51 MB of the compressed data with Tar

Table 2: Average overheads of popular applications with RMA. Launching application is the worst case metric to RMA because it includes the overheads of initiating key shares although it is one time cost. The overhead of memory attestation varies depending on workloads, from -2.8% when compressing files to 15.9% when launching an email client.

7.3 Performance Break-down

We also measured how long does it take to proceed each stage of the RMA protocol with our prototype implementation. We denoted the amount of data that need 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 (see Table 3) 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.

8 Conclusions

We initiated formal treatment of the problem of remotely testing if a memory (such as heap) has been infected with a malicious code, without relying on trusted hardware. Our work combines solid theoretical foundations of provable security with the systems expertise of the application. Towards this goal, we study the practical threats and formalize the security definition for remote memory attestation protocols. Our security definition

Seq	Role	Task Description	Data	Time
	Verifier	Generating public key and secret key	-	0.291s
	Prover	Generating `xor`-ed of all shared key (482 shares on 128 MB memory)	-	0.034s
	Verifier	`INIT` is sending to wrapper to get the `xor`-ed of all shares.	-	0.4ms
❶	Prover	Sending `xor`-ed of all shared keys.	16 bytes	0.2ms
	Verifier	Receiving 16 bytes from the wrapper.	-	0.5ms
	Verifier	Sending a challenge to wrapper.	-	0.016ms
❷	Prover	Receiving a message from verifier	12 bytes	0.012ms
	Prover	Encrypting message	-	0.409s
❸	Prover	Sending cipher to verifier	384-396 bytes	0.007s
❹	Verifier	Receiving cipher from the wrapper	384-396 bytes	0.357s
	Verifier	Decrypting cipher	-	0.234s

Table 3: Time and data transferred when performing remote memory attestation of an application having 128 MB memory, which includes 482 number of shares.

is very general and can be easily customized for various settings classes of attacks. We propose two RMA protocol constructions to prevent limited, but still practical classes of attacks, when the adversary overwrites a consecutive region of memory to compromise a control-sensitive data structure. The first protocol uses a hash function and a simple XOR-based secret sharing scheme. We prove its security in the random oracle model. For stronger security guarantees in the standard model, we propose a protocol based on a recently proposed public key encryption scheme, and the same XOR-based secret sharing scheme. We prove security of this construction under the standard computational assumptions. We demonstrate feasibility of our design with implementation results.

References

[1] Michel Abdalla, Fabrice Benhamouda, and David Pointcheval. Public-key encryption indistinguishable under plaintext-checkable attacks. In Jonathan Katz, editor, *Public-Key Cryptography - PKC 2015 - 18th IACR International Conference on Practice and Theory in Public-Key Cryptography, Gaithersburg, MD, USA, March 30 - April 1, 2015, 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] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, pages 1–8, Tel-Aviv, Israel, 2013.

[4] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian 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.

[5] Arash Baratloo, Navjot Singh, Timothy K Tsai, et al. Transparent run-time defense against stack-smashing attacks. In *USENIX Annual Technical Conference, General Track*, pages 251–262, 2000.

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

[7] Mihir Bellare, David Cash, and Rachel Miller. Cryptography secure against related-key attacks and tampering. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 486–503. Springer, 2011.

[8] Mihir Bellare and Tadayoshi Kohno. A theoretical treatment of related-key attacks: Rka-prps, rka-prfs, and applications. In *Advances in CryptologyâĂŤEUROCRYPT 2003*, pages 491–506. Springer, 2003.

[9] Mihir Bellare, Kenneth G. Paterson, and Susan Thomson. RKA security beyond the linear barrier: Ibe, encryption and signatures. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 331–348. Springer, 2012.

[10] Mihir Bellare and Phillip 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.

[11] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006.

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

[13] Rishiraj Bhattacharyya and Arnab Roy. Secure message authentication against related-key attack. In Shiho Moriai, editor, *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, volume 8424 of *Lecture Notes in Computer Science*, pages 305–324. Springer, 2013.

[14] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *Journal of the ACM (JACM)*, 51(4):557–594, 2004.

[15] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio 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, pages 400–409. ACM, 2009.

[16] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian 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.

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

[18] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. What if you can't trust your network card? In *Recent Advances in Intrusion Detection*, pages 378–397. Springer, 2011.

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

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

[21] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In Don Coppersmith, editor, *Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August 27-31, 1995, Proceedings*, volume 963 of *Lecture Notes in Computer Science*, pages 339–352. Springer, 1995.

[22] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan 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)*, pages 1–8, Tel-Aviv, Israel, 2013.

[23] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 265–278, 2013.

[24] Markus 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.

[25] Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin, and John Butterworth. New results for timing-based attestation. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 239–253. IEEE, 2012.

[26] Yanlin Li, Jonathan M McCune, and Adrian Perrig. Sbap: Software-based attestation for peripherals. In *Trust and Trustworthy Computing*, pages 16–29. Springer, 2010.

[27] Yanlin Li, Jonathan M McCune, and Adrian 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.

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

[29] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, 2010.

[30] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, New York, NY, USA, 2008. ACM.

[31] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday 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)*, pages 1–8, Tel-Aviv, Israel, 2013.

[32] Nick Nikiforakis, Frank Piessens, and Wouter Joosen. Heapsentry: Kernel-assisted protection against heap overflows. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 10th International Conference, DIMVA 2013, Berlin, Germany, July 18-19, 2013. Proceedings*, pages 177–196, 2013.

[33] Tatsuaki Okamoto and David Pointcheval. REACT: rapid enhanced-security asymmetric cryptosystem transform. In David Naccache, editor, *Topics in Cryptology - CT-RSA 2001, The Cryptographer's Track at RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings*, volume 2020 of *Lecture Notes in Computer Science*, pages 159–175. Springer, 2001.

[34] William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Run-time detection of heap-based overflows. In *Proceedings of the 17th USENIX Conference on System Administration*, LISA '03, pages 51–60, 2003.

[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, 2012.

[36] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep 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, pages 1–16, 2005.

[37] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. Swatt: Software-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 272–282. IEEE, 2004.

[38] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, 2004.

[39] Victor Shoup. Sequences of games: A tool for taming complexity in security proofs, 2004.

[40] Hoeteck Wee. Public key encryption against related key attacks. In Marc Fischlin, Johannes A. Buchmann, and Mark Manulis, editors, *Public Key Cryptography - PKC 2012 - 15th International Conference on Practice and Theory in Public Key Cryptography, Darmstadt, Germany, May 21-23, 2012. Proceedings*, volume 7293 of *Lecture Notes in Computer Science*, pages 262–279. Springer, 2012.

[41] Yves Younan, Wouter Joosen, and Frank 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.