Provable Virus Detection: Using the Uncertainty Principle to Protect Against Malware*

[Extended Abstract]

Richard J. Lipton
Georgia Institute of Technology
rjl@cc.gatech.edu

Rafail Ostrovsky
UCLA
rafail@cs.ucla.edu

Vassilis Zikas*
ETH Zurich
vzikas@inf.ethz.ch

ABSTRACT

Protecting software from malware injection is the holy grail of modern computer security. Despite intensive efforts by the scientific and engineering community, the number of successful attacks continues to increase.

We have a breakthrough novel approach to provably detect malware injection. The key idea is to use the very insertion of the malware itself to allow for the systems to detect it. This is, in our opinion, close in spirit to the famous Heisenberg Uncertainty Principle. The attackers, no matter how clever, no matter when or how they insert their malware, change the state of the system they are attacking. This fundamental idea is a game changer. And our system does not rely on heuristics; instead, our scheme enjoys the unique property that it is proved secure in a formal and precise mathematical sense and with minimal and realistic CPU modification achieves strong provable security guarantees. Thus, we anticipate our system and formal mathematical security treatment to open new directions in software protection.

General Terms

Malware Detection, Provable Security, Attestation

1. INTRODUCTION

Protecting software from malware injection is a major goal of computer security. In this work we suggest a novel provably secure and practically efficient paradigm for software protection against arbitrary malicious code injection.

Our system has the following desirable properties: 1) it requires minor to no changes to the CPU specification to provably defend against all injection attacks that can not read the code before they inject its software; 2) it only affects the performance of the program by a modest amount (or not at all, given slightly more powerful CPU computation). The first property implies that it can already be used on today's computers. The second property means that our system is practical, since performance is critical in most applications; this is obvious even in its simplified form presented here, where various optimizations have been avoided for the sake of clarity in the presentation. Finally, we allow attacks both spatial and temporal freedom: the attack can occur at any time during the execution and on any part of the memory. It can attack code as well as data.

Our solutions are practically efficient and, importantly, they are accompanied by rigorous mathematical proofs that any injection will be caught with arbitrary high probability. For our security proofs we provide a formal cryptographic model tailored to detect malware injection in modern computers; we envision this model as the basis for fruitful research on provable secure detection and prevention of malware leading into the next generation of secure systems.

1.1 Overview of our Approach

So how can we detect an intrusion that we have never seen before? How can we arrange that any injection of malware into our program, no matter how clever, will be detected? The answer is that we will hide a secret in our program in a way that ensures that with very high probability, any injection by an adversary must destroy the secret. Once this secret is destroyed, the adversary may indeed be able to take over the program, but will be quickly detected. We note that current systems may fail to detect such an attack forever—we are able to detect it in seconds.

A point: We can imagine situations where even such a swift detection of an attack is not sufficient to stop all potential harm. An attacker could, in some situations, do immense damage even if he is in control of a program for only a fraction of a second. Furthermore, our approach is not targeted towards fixing buggy software. And we do not necessarily detect malware-including software which might be (unknowingly) installed by the user. Rather, our detection offer protection against "unauthorized" injection of malicious code, e.g., through buffer overflows. But the ability to detect any attack, new, old, clever, or not, is an immense improvement over the current state-of-the-art.

So how can we do this? Let W be a program that we wish to protect from malware. We recompile W to $\widetilde{W}$. The idea is that W and $\widetilde{W}$ must compute the same thing, run in about the same time, and yet $\widetilde{W}$ must be able to detect itself against the insertion of malware. An obvious idea is to have $\widetilde{W}$ periodically check to see if its code or data have been maliciously changed. The trouble with this approach is that the attacker could easily inject his own code, take over the checker, and thereby disable the checking. In short: who checks the checker?

Here is how we proceed. The protected program $\widetilde{W}$ operates normally most of the time. Periodically it is challenged by another machine to prove that it has a secret key K. This key is known only to $\widetilde{W}$ and not to the attacker. If $\widetilde{W}$ has not been attacked, then it simply uses the key K to an-

*A patent is pending (Application Nr. 62/054,160).
*Research done in part while the author was at UCLA.

swer the challenge and thus proves that it is still operating properly.

The issue is what happens if $\widetilde{W}$ has been attacked and some code has been injected into it. We can arrange $\widetilde{W}$ so that no matter how the injection of code has happened the key K is lost. The attacker's very injection will have changed the state of $\widetilde{W}$ so that it is now in a state that no longer knows the key K. This is the analog of the Heisenberg Uncertainty Principle: the attacker has damaged the state of $\widetilde{W}$ so that the key has been destroyed.

Making the attack destroy the key is the central idea here. If $\widetilde{W}$ simply stores K somewhere in memory, the attacker will take over the program, look around for the key, and likely find the key; thereby defeating the protection, since now it can answer the periodic challenges just like $\widetilde{W}$ could. This would be a disaster.

To resolve the above issue we distribute the key K all through memory by using what is called *secret sharing* [41]. This is a standard method in cryptography that breaks a small key, like K, into many small pieces, called shares. These pieces are cleverly placed throughout all of $\widetilde{W}$'s memory. Our secret sharing allows K to be reconstructed only if all the pieces are left untouched—if any are changed in any way, then it is impossible to reconstruct the key. Obviously, if there has been no attack, then in normal operation $\widetilde{W}$ can reconstruct the key from the pieces and answer the challenges. However, if the pieces are not all intact, then $\widetilde{W}$ will be unable to answer the next challenge.

This is the high level idea of our method: hide the key via secret sharing, and rely on the attacker to destroy at least one share of the key. This destruction is irreversible and makes the attack fail the next challenge.

There is one additional point that we must mention. We must arrange that $\widetilde{W}$ can be attacked at anytime, including when it has collected all the shares and reconstructed the key K. This is a very dangerous time, since if $\widetilde{W}$ is attacked at this moment the fact that shares are destroyed does not matter. The attacker can simply use the reconstructed key K. We avoid such *time-of-check-time-of-use* (TOCTOU) attacks [36] by actually using two keys K_1 and K_2 in tandem. Both are stored via secret sharing as before, but now one must have both in order to answer the challenges. We arrange that $\widetilde{W}$ only reconstructs one key at a time and this means that an attacker must destroy either or bot of the keys. This handles the above dangerous situation and makes our method provably secure.

Tying up the last loose end, we treat the case of very small viruses—i.e., ones that consist of a single bit or just a few bits—and viruses that know (or guess) the exact memory locations of key-shares in advance and therefore might not need to overwrite them. We armor our system to defend even against such tiny/informed viruses by employing an additional lightweight cryptographic primitive, namely a *message authentication code* (in short, MAC). A MAC authenticates a value using a random key, so that an attacker without access to the key cannot change the value without being detected.

We use the MACs in a straightforward and efficient manner: we authenticate each word in W using the key-shares (also) as MAC keys, and require that for any word that is loaded from the random access memory, the CPU verifies the corresponding MAC before further processing the word. Thus, if the MAC does not check out we detect it (and immediately destroy the share), and if the adversary changes the MAC key, he loses one of the shares of one of the two secrets, and we detect that as well.

We enforce the above checks by assuming a modified CPU architecture (with only very small amount of additional computation, so the CPU power consumption is not affected.) More specifically, to get the most out of the MAC functionality we need that when the CPU executes the program, it verifies authenticity of the words it loads from the memory. That is, the load operation of the CPU loads a block of several (four or five) words—this is already the case in most modern CPUs—including the program word, the MAC-tag and the corresponding keys, and check with every load instruction that the MAC verifies before further processing the word. Importantly, our MAC uses just one field addition and one field multiplication per authentication/verification; the circuits required to perform these operation are actually trivial compared to the modern CPU complexity and are part of many existing CPU specifications.

But to obtain security against any injection, we need one more trick: instead of using the actual key-shares in the generation/verification of the MAC tag, we use their hash-values. This ensures that the virus cannot manipulate the keys and forge a consistent MAC unless he overwrites a large part of them. We anticipate that given our proposed system's impact on security, such functionality will be included in the next generations of CPUs [35].

1.2 Related Literature

The literature on modelling, generalizing, and providing defense mechanisms against malware-injection attacks is vast. Due to its urgency and importance, the problem has been intensively researched. In what follows we provide a brief overview of the basic research directions that have been explored and refer to the full version for a more detailed exposition and a complete list of references.

Common Security Countermeasures. The traditional and most applicable countermeasures to prevent malware attacks are antivirus technologies along with approaches directed at blocking the path an attacker uses to inject its malware, e.g., buffer-overflows [3, 9]. Other defenses monitor the system calls that the program makes—e.g., its interfaces to the operating system—to detect abnormal behavior [20, 19, 18]. Although the above countermeasures are often successful in patching known vulnerabilities, they are typically *ad-hoc* and/or targeted to a single or a small class of attacks, thus leaving a large part of the system exposed to unforeseen or zero-day attacks.

At a more theoretical level, the *software diversification* approach uses randomization to ensure that each system executes a unique (or even several unique) copies of the software, so that an attack which succeeds against some system will most likely fail against other systems (or even when launched twice against the same system), e.g., [8, 22, 24, 42, 10, 27]. This idea is inspired by biological systems, where diversification is a prominent venue for resiliency. For example *address obfuscation* randomizes the locations of program data and instructions so that the attacker is most likely to follow an invalid computation path and thus provoke a crash or fault [24, 4].

It should be noted that the above approaches are typically *heuristics*, and there is no formal proof that they are ei-

ther efficient or effective; rather their performance and accuracy are typically experimentally validated. (Many of these experiments indicate a potentially undesirable trade-off between security and accuracy of the respective method.)

Attestation. Closer related to our goals is the literature on verifying the authenticity/integrity of the internal state of computing devices to confirm that they have not been attacked by malware, a mechanism usually referred to as *attestation*. A rough taxonomy divides this literature in three general categories, namely *hardware-based attestation* [2, 7, 15, 33], *software-based attestation* [31, 39, 38, 1, 28, 29], and *remote attestation* [7, 15, 33, 21]. Due to the similarity in the goals of attestation with our system, we present a detailed overview of the corresponding literature in the full version of this work. As a general note, we stress that similarly to the aforementioned defense mechanism, attestation schemes do not come with a formal proof of security or even a theoretical security model. In contrast, our scheme is not only backed by a formal mathematical proof, but has several additional advantages with respect to existing attestation schemes, which are highlighted in Section 1.4.

Cryptographic Solutions. On the most theoretical side of cybersecurity, cryptography provides solutions whose security is backed by rigorous mathematical proof that typically reduce hardness of breaking the scheme to hardness of solving a mathematical problem, e.g., factoring. Unfortunately, with only a few exceptions, e.g., the recent work by Dachman-Soled et al. [11] and by Faust et al. [17] which build on special *non-malleable codes* [14], the so-called *proofs of retrievability* [30] which are aimed in verifying integrity of stored static data, and the recent works on *program obfuscation,* e.g., [23, 37], the cryptographic literature has mostly overlooked the problem of malicious code injection. Moreover, in contrast to the security-engineering research, cryptographic solutions are often inefficient and/or adopt a too-abstract model of computation which makes them inapplicable or impractical for today's systems.

1.3 Our Approach in More Detail

We put forward and formally define the notion of a *virus detection scheme* (in short, VDS) which compiles any given program W (and its data) into a new secured program $\widetilde{W}$ that performs the same computation as W but allows us to detect any virus injected in the memory at any point of the execution path. The detection is done via a provably secure challenge-response mechanism between the machine executing the (compiled) software and a verifying external device. Importantly, we insist that the verification algorithm be very simple, and in particular that it be executed by a very lightweight device (Our constructions require that the verifier only does encryption and compare strings for equality.) Thus the role of the verifier can be, for example, played by the user with a smartphone, or by a compact and simple, intrusion-free hardware module that could even be part of the CPU.[1] Moreover, the verification uses public key techniques, where the verifier knows the public key. This restricts the attack surface to the machine executing the secured code, as compromising the verifiers privacy gives an attacker no advantage in breaking our scheme.

[1]Unlike software-based attestation techniques, the verification in our scheme can be done even over an insecure network, e.g. the Internet.

We provide our formal cryptographic definitions of VDS security based on the well-studied Random Access Machine (RAM) model, slightly adapted to be closer to an abstract version of a modern computer following the *von Neumann architecture*. The suggested model corresponds to a simplistic *closed-system* abstraction of software execution, where the execution of code is done by the software's code and data loaded (initially) in the random access memory (RMEM) which through the execution only communication with the CPU.[2] We provide a formal and generic definition of malware injection in this model, which is motivated by how modern computer systems might become infected by a virus.

In more detail, a secure VDS is described as follows. The software to be executed is compiled, prior to being loaded onto the memory, to a secure version. Importantly, this compilation does not need the source code but merely the binary (machine-language) code of the program (provided that it is non-self-modifying); thus, the compilation can be performed by the program vendor (this might allow for further efficiency optimizations) or by the user without requiring any reverse engineering. The compiler, in addition to the (binary code) of the program and its data, uses a randomly chosen *compilation key* K. To check/verify that the software running on a system has not been attacked by a virus (or to detect such an attack in case it has occurred), the following *detection process* is executed. The user issues a challenge c that depends on the compilation key K and the system is required to produce a reply which passes a specified verification test. As already discussed, we arrange things in such a way that the verifier does not need to know the compilation key, and he can just know some public information on it. Formally, this is done by the use of public key cryptography, where the compilation key is the secret key, and the information held by the verifier, which we refer to as the *verification key*, is the corresponding public key. The security of the VDS requires that if the RAM has not been attacked then it can always reply to the challenge in an accepting manner (verification correctness); otherwise, any reply will be rejected with high probability (security).

We provide concrete instantiations of practical VDSs which, depending on the assumptions about the CPU they are executed on, can achieve from a reasonably strong security (namely protection against moderate-sized virus) to the ultimate security guarantee (namely security against viruses that are arbitrary small and are injected on locations that depend on the location of the key shares). Informally, we prove the following result for our VDSs.

Theorem (informal) *Under standard complexity assumptions our VDSs compile any non-self-modifying software W into a secure version that detects any malware injection with very high probability.*

Our schemes are independent of the virus code, are platform-independent, and might require only small modifications to the common CPU architecture (in particular, they do not assume tamper-resilient hardware); and, with appropriate optimizations we can make it that they only affect the performance of the executed software by a practically unnoticeable amount. However, as this is the first work

[2]In particular, the current theoretical model does not address how the data is exchanged with secondary storage devices, e.g., flash memory. It is part of ongoing research to extend this model to more complicated architectures.

introducing VDSs, various performance optimizations have been sacrificed for the sake of presentation clarity. As a useful side effect, our scheme is even able to detect hardware errors, e.g., random bit-flips in the memory; as demonstrated in [6], such errors might have important consequences on the security of the executed software.

For the construction of our VDSs we devise a general methodology. We start by constructing a scheme satisfying a weaker notion of security which ensures that the adversary is caught if the response is computed by an external (trusted) device that is given access to the state of the attacked system. We give a formal definition of this weaker notion which depending on the application and the desired security level might already be satisfactory. We then proceed and gradually modify our weaker scheme to ensure the compiled program executes the detection by itself. The security of our weakest scheme is information theoretic (it is based on the perfect privacy of one-time pad encryption), whereas the proof of the more secure scheme requires a leakage-resilient encryption algorithm [13] and a secure hash function.

Our last, ultimately secure VDS uses the following simple MAC algorithm. To authenticate a word w we use two keys K_1 and K_2, and compute a MAC tag as $t = w \cdot K_1 + K_2$, where w, K_1, and K_2 are interpreted as elements of an appropriate large arithmetic field. Observe that such arithmetic operations are implemented in hardware in existing CPUs. We assume, however, that the executing CPU has a slightly extended instruction set which, informally, has each "load" instruction, i.e., read-from-memory instruction, check that the MAC is correct before it loads the word on the CPU registers. We stress that this only needs the architecture to provide us with a very simple extra piece of microcode, but by engineering the CPU's hardware in a smart way so that these operations are done on the circuit level, we expect to be able to reduce the effect of our compiler to a barely noticeable slowdown. Fortunately, the new architecture that Intel recently announced promises to embed such microcode in its next-generation microchips [35].

1.4 Comparison to Existing Approaches

In Section 1.2 above we discussed main directions in existing malware detection and protection literature. Here, we highlight the advantages of our scheme. As already mentioned, a major advantage with respect to existing works is that our scheme comes with a mathematical security proof, i.e., its security does not rely on simulations. In what follows we discuss some of our scheme's basic advantages with respect to existing methods:

State-independent verification. The verifier does not need to know the state in which the executed software is at the point of verification; in fact the state of the verifier is independent of the software's computation. Thus our scheme can protect *both static program and its data* and has the potential to be extended to classes of dynamic (self-modifying) programs.

Mitigating denial of service. Existing attestation techniques, whether remote or software-based, require the host to pause computation, as the state of the machine should not change during the execution. (If the state changes the checksum will fail.) This allows for trivial denial-of-service attacks. Instead, our verification does not require the host to pause its execution of the software, nor does it require disallowing interrupts thus mitigating denial of service.

No need for timing or reliable communication. Software-based attestation schemes typically require accurate timing and a reliable connection to the receiver. (The latter requirement is necessary even in remote attestation schemes.) In contrast, our scheme allows for remote verification/attestation without the need for any of the above properties. Indeed, even when taking full control over the verification link, an attacker can respond correctly only if the host has not been compromised.

No secrets held by the verifier. By using public key cryptography, we restrict the attack surface to be only the host. In particular, even if the entire environment of the system which is verified is under the control of the adversary, it will be provably infeasible to come up with a valid reply to a challenge, as the only entity that can recover the private key is the host itself. This is particularly useful in settings where the system is a *service provider, e.g., a cloud server*, and might be verified over the Internet by different users. In contrast, in existing remote or software attestation schemes, knowledge of the long-term or short-term key can trivially compromise the schemes security.

A Single Realistic Assumption. Our scheme is secure under the assumption that the attacker does not get to see information on the key-shares without injecting its code first. This is for example the case with many injection venues, e.g., buffer-overflows. Note that this assumption suffices to remove all past assumptions in remote or software attestation, i.e., minimality and non-parallelizability of code, complete knowledge of the hardware, etc [21].

1.5 Outline

The remainder of the paper is organized as follows: In Section 2 we introduce some useful notation and terminology; subsequently, in Section 3 we describe the model of execution and malware injection which is used to introduce Virus Detection Schemes (VDS) and describe their security definition in Section 4. In Sections 5 and 6 we describe our secure VDS instantiations: We start (in Section 5) with a scheme that is secure against moderately-sized (at least three words long) viruses injected in continuous memory location and poses no architectural restrictions on the system it is executed on (thus it can be applied on modern as well as legacy systems); our system is strengthened in Section 5 to tolerate injection of arbitrary small viruses and remove the continuous injection assumption, under minimal and realistic modifications on the systems CPU. This manuscript is an extended abstract of our work; detailed definitions and formal proofs of our statements can be found in the full version of this work available from the authors.

2. PRELIMINARIES AND NOTATION

Throughout the paper we assume an (often implicit) security parameter denoted as k. For a finite set S we denote by $s \xleftarrow{\$} S$ the operation of choosing s from S uniformly at random. For a randomized algorithm $\mathtt{B}$ we denote by $\mathtt{B}(x;r)$ the output of $\mathtt{B}$ on input x and random coins r. To avoid always explicitly writing the coins r, we shall denote by $y \xleftarrow{\$} \mathtt{B}(x)$ the operation of running $\mathtt{B}$ on input x (and uniformly random coins) and storing the output on variable y. We use the standard definition of *negligible* and *overwhelming* (e.g., see [25]). For a number $n \in \mathbb{N}$, we denote by $[n]$

the set $[n] = \{1, \ldots, n\}$ and by 0^n (resp. 1^n) the all-zero (resp. all-ones) string of length n. Finally, for strings x and y we denote their concatenation by $x||y$. In slight abuse of notation, we at times use the vector notation for denoting concatenation, i.e., write (x, y) instead of $x||y$.

Secret Sharing. A perfect m-out-of-m secret sharing scheme allows a dealer to split a secret s into m pieces $s_1, \ldots, s_m$, called the *shares* so that someone who holds all m shares can recover the secret (correctness), but any $m-1$ shares give no information on the secret (privacy). In this work we use m-out-of-m sharings of strings $s \in S = \{0,1\}^n$, where $n \in \mathbb{N}$. A simple construction of such a scheme has the dealer choose all s_i's uniformly at random with the restriction that $s = \bigoplus_{i=1}^{m} s_i$. We refer to the above sharing as the *XOR-sharing* and denote it as $\langle s \rangle = (s_1, \ldots, s_m)$.

3. THE MODEL

In this section we provide an abstract specification of our model of computation and a model for malware injection. We use as basis the well known Random Access Machine (RAM) model but slightly adapt it to be closer to an abstract version of a modern computer following the *von Neumann architecture*. In a nutshell, this modification consist of assuming that both the program and the data are written on the RAM's random access memory[3] which is polynomially bounded. Along the way we also specify some terminology which draws parallels between our theoretical model and modern computers' specification.

A RAM $\mathcal{R}$ consist of two components: A *Random Access Memory (in short RMEM)*[4] and a *Central Processing Unit (in short CPU)*. The memory RMEM is modeled as an array of $m = poly(k)$ registers each of which can store an L-bit string—*word*—where we assume that L is linear in the security parameter. The CPU has a much smaller set of registers—in this work we assume that the total amount of storage of the CPU is linear in the security parameter k—an instruction set $\mathcal{I}$ that defines which operations can be performed on the registers, and how data are loaded-to/output-from the CPU. The CPU registers include a read-only input-register and output register which correspond to its interface with its environment (the user), and a *program counter* `pc` which stores the location in the memory that the will be read in the next CPU cycle. Each register is assigned a unique address and can store an L-bit word.

The RMEM and the CPU communicate in *fetch and execute cycles, aka CPU cycles*, where the number of CPU cycles is the default complexity measure of a RAM.[5] We at times refer to a CPU cycle as a *round* in the RAM execution. In each round, the CPU accesses the RMEM to read (load) a word to some of its registers, performs some basic operation from $\mathcal{I}$ on its local registers, e.g., adding two registers and storing their output on a third register, and (potentially) writes some word (the contents of some register) to a location in the memory RMEM. The location in RMEM from where the next word will be read is stored in a special integer-valued register denoted as `pc` (usually called the *program counter*) which unless the CPU processes a "jump" instruction (see below) is incremented by one in each cycle. Although modern CPUs are able to execute several instructions per round, here we assume for simplicity that only one instruction is executed in each round. In the following we provide some details on the above components.

We represent the memory RMEM as an array `MEM` of $m = |\texttt{MEM}|$ words, where for $i \in \{0, 1, \ldots, m\}$ we denote by $\texttt{MEM}[i]$ the ith word, i.e., the contents of the i-th register. Similarly, the CPU registers are modeled as an array `REG` of words, where we denote by $\texttt{REG}[i]$ the content of the ith register.

We denote a CPU by the pair $\mathcal{C} = (\texttt{REG}, \mathcal{I})$ of the vector `REG` of registers and the instruction set $\mathcal{I}$. As syntactic sugar, we denote a RAM with CPU $\mathcal{C}$ and RMEM `MEM` as $\mathcal{R} = (\mathcal{C}, \texttt{MEM})$. The *state* of a $\mathcal{R}$ at any point in the protocol execution is the vector $(\texttt{REG}, \texttt{MEM})$ including the current contents of all its CPU and RMEM registers.

To allow for asymptotic security definitions—where the word size, the size of the CPU (i.e., number of its registers), and the size of the memory depend on the security parameter—we often consider a family of RAM's, $\mathcal{R} = \{\mathcal{R}_k\}_{k \in \mathbb{N}}$ with $\mathcal{R}_k = (\mathcal{C}_k = (\texttt{REG}_k, \mathcal{I}_k), \{\texttt{MEM}_k\})$. (The size of words processed by $\mathcal{R}_k$ is k). The RAM families considered in this work have the following property: The instruction set $\mathcal{I}_k$ is the same for all value of the security parameter k. In particular, we assume that all elements of a RAM family, have the same constant number $c = \mathcal{O}(1)$ of CPU registers, and there is some set of instructions $\mathcal{I}$ that is defined over strings of arbitrary size such that $\mathcal{I}_k$ is the same as $\mathcal{I}$ applied on k-long words.

Software Execution and Virus Injection. A program to be executed on a RAM $\mathcal{R} = (\mathcal{C}, \texttt{MEM})$ is described as a vector $W = (w_0, \ldots, w_{n-1}) \in (\{0,1\}^L)^n$ of words that might be instructions, addresses, or program data. To avoid confusion, we refer to such a vector including the (binary of a) software and its corresponding data as a *binary* or a *programming for* $\mathcal{R}$. By convention, whenever, for a RAM family $\mathcal{R}$ we say that W is programming for $\mathcal{R}$, we mean that W is a programming for the element $\mathcal{R}_k \in \mathcal{R}$ with register size as long as the word size of W and instruction seat that includes all the instructions used by W.

The execution of a program proceeds as follows: The programming W is loaded onto the memory `MEM`. Unless stated otherwise, we assume that W is loaded sequentially on the first $n = |W|$ locations of `MEM` and locations j of `MEM` with $j \geq |W|$ are filled with (`no_op`) instructions. The user might give input(s) to $\mathcal{R}$ by writing them on its input register in a sequential (aka write once) manner, i.e., a new input is written (appended) next to the last previous input. Once the binary is loaded (and, potentially, input has been written on the input register) the RAM starts its execution by fetching the word of the RMEM which the program counter points to, i.e., $\texttt{MEM}[\texttt{pc}]$. Unless stated otherwise, at the beginning of the computation `pc` is initiated to 0 (i.e., points to the first memory location). The RAM executes the program in CPU cycles as sketched above. We assume that the RAM is reactive, i.e., any new input written (appended) on its input register, makes the RAM resume its computation (even if it had halted with output).

A CPU is *complete* (also referred to as *universal*) if given sufficient (but polynomial) random access memory it can

[3]In the literature, a RAM with this modification is usually called a *Random Access Stored-Program* machine [16].

[4]We use RMEM instead of the more standard "RAM" to refer to random access memory as here RAM refers to Random Access Machine.

[5]In reality, this communication is done through the CPU bus; however for our treatment we do not need to formally specify how this communication is implemented.

perform any efficient deterministic computation. We at times refer to a RAM (family) with a complete CPU as a complete RAM (family). An example of a complete CPU is one which has three registers (each of size k) and can, in addition to communicating with its RMEM, compute arithmetic operations on the contents of any two of these registers (treated as representations of elements from an appropriate finite field) and store it on the third. However, modern CPUs have many more registers and might perform several complicated instructions in a single round.

Modeling Virus Attacks. A virus attacks a RAM by injecting its code on selected locations of the memory RMEM. For simplicity, we discuss here viruses that inject sequences of full words but our definitions are easily extended to apply also to viruses whose length (in bit) is not a multiple of the word length L. (In fact, in Section 6 we show how to protect against such viruses.) More formally, an ℓ-word virus is modelled as a tuple $\mathbf{v} = (\vec{\alpha}, W) = ((\alpha_0, \ldots, \alpha_{\ell-1}), (w_0, \ldots, w_{\ell-1}))$, where each $\alpha_i \in \vec{\alpha}$ is a location (address) in the memory and each $w_i \in W$ is a word. The effect of injecting a virus $\mathbf{v}$ into a RAM $\mathcal{R} = (\mathcal{C}, \mathtt{MEM})$, is to have, for each $\alpha_i \in \vec{\alpha}$, the register $\mathtt{MEM}[\alpha_i]$ (over)written with w_i. We say that the $\mathbf{v}$ is valid for $\mathcal{R}$ if the following properties hold: (1) $\alpha_i \neq \alpha_j$ for every $\alpha_i, \alpha_j \in \vec{\alpha}$, and (2) $\alpha_i \in \{0, \ldots, |\mathtt{MEM}| - 1\}$ for every $\alpha_i \in \vec{\alpha}$. Furthermore, we say that $\mathbf{v}$ is *non-empty* if $|\vec{\alpha}| > 0$.

We make no external security assumption, e.g., existence of secure hardware, about the CPU; thus, for example, an "intelligent" virus is allowed to take full control of the CPU, i.e., read and (over)write all its registers, while it is being executed. But we do not allow the virus to inject itself on the CPU registers. This is justified by the fact that during the software execution, the CPU communicates only with the RAM and the input/output devices. Finally, we point out that some of our theorems assume a *continuous injection model,* i.e., that the virus is injected as a single chunk in continuous memory locations.

4. VIRUS DETECTION SCHEMES

In this section we define *virus detection schemes* and provide the relevant security definitions.

Definition 1. A *virus detection scheme (VDS)* $\mathcal{V}$ consists of five (potentially) randomized algorithms, i.e., $\mathcal{V} = (\mathtt{Gen}, \mathtt{Comp}, \mathtt{Chal}, \mathtt{Resp}, \mathtt{Ver})$, defined as follows:[6]

- `Gen` is a key-generation algorithm; it computes a pair (compilation-key, verification-key), i.e., $(K_c, K_v) \xleftarrow{\$} \mathtt{Gen}$.[7]
- `Comp` on input the description $\mathcal{R} = \{(\mathcal{C}_k, \mathtt{MEM}_k)\}_{k\in\mathbb{N}}$ of a RAM family,[8] a programming W for $\mathcal{R}_k$, and a compilation key K_c, `Comp` outputs a *secure programming* $\widetilde{W}$ for $\mathcal{R}_k$; i.e., $\widetilde{W} \xleftarrow{\$} \mathtt{Comp}(\mathcal{R}_k, W, K_c)$.
- `Chal` on input a verification key K_v, and a string $z \in Inp_{\mathtt{Chal}} \subseteq \{0,1\}^{poly(k)}$, `Chal` outputs a challenge string $c \in Out_{\mathtt{Chal}}$, where $Out_{\mathtt{Chal}} \subseteq \{0,1\}^{poly(k)}$ denotes the output domain of algorithm `Chal`; i.e., $c \xleftarrow{\$} \mathtt{Chal}(z, K_v)$.
- `Resp` on input a string $c \in Out_{\mathtt{Chal}}$ and a polynomial programming $\widetilde{W}$, `Resp` outputs a string $y \in Out_{\mathtt{Resp}}$, where $Out_{\mathtt{Resp}} \subseteq \{0,1\}^{poly(k)}$ denotes the output domain of algorithms `Resp`; i.e., $y \xleftarrow{\$} \mathtt{Resp}(c, \widetilde{W})$.
- `Ver` on input a verification key K_v, a message $z \in Inp_{\mathtt{Chal}}$, a challenge $c \in Out_{\mathtt{Chal}}$ and a response $y \in Out_{\mathtt{Resp}}$, `Ver` outputs a bit $b \in \{0,1\}$; i.e., $b \xleftarrow{\$} \mathtt{Ver}(K_v, z, c, y)$. We say that `Ver` accepts if $b = 1$.

In the following we sketch the security properties which a VDS should satisfy. Due to space limitation, we restrict ourselves to high-level descriptions and refer to the full version of this paper for a detailed description of the random experiments underlying these definitions.

- The first property is *verification correctness* which, intuitively, guarantees that if the RAM has not been attacked, then the reply to the challenge is accepting (except with negligible probability).
- The second property is *compilation correctness*, which intuitively ensures that the compiled programming $\widetilde{W}$ performs the same computation (on $\mathcal{R}$) as the original programming W. This means that on the same input(s) from the user the following properties hold: (1) $\mathcal{R}$ executing $\widetilde{W}$ produces the same output sequence as $\mathcal{R}$ executing W, and (2) there exists an efficient transformation which maps executions of $\mathcal{R}$ programmed with $\widetilde{W}$ onto executions of $\mathcal{R}$ programmed with W, such that the contents of the memory `MEM` at any point of the the execution of $\mathcal{R}$ with programming $\widetilde{W}$ are efficiently mapped to contents of `MEM` at a corresponding point in the execution of $\mathcal{R}$ with programming W.[9]
- In an application of a VDS, the verifier inputs the challenge at a point of his choice and checks that the reply verifies according to the predicate `Ver`. Thus the last property we require from a VDS is *self-responsiveness*: the secured programming $\widetilde{W}$ includes code that on some special input emulates algorithm `Resp` on the RAM it executes. More concretely, the requirement here is that upon receiving a special input message $x' = (\mathtt{check}, c)$ the next output of the RAM equals the output of `Resp` on input c with overwhelming probability.

Virus Vulnerability as a Security Game. The aforementioned properties, specify the behavior of software protected by a VDS when it is not attacked by malware. We next define invulnerability of a VDS to malware-injection attack via a security game between an adversary `Adv` who aims to inject a virus on a RAM, and a challenger `Ch` who aims to detect it. Informally, the security definition requires that `Adv` cannot inject a virus without being detected, except with sufficiently low probability.

At a high-level, the security game, denoted by $\mathcal{G}_{\mathtt{VDS}}^{\mathcal{R},\mathcal{V},W}$, proceeds as follows: The challenger `Ch` runs the key-generation algorithm to obtain a key-pair (K_c, K_v), and

[6] All five algorithms below take as an additional input the security parameter k, which is omitted for compactness.

[7] Note that if we instantiate the VDS with symmetric-key cryptography then $K_c = K_v$.

[8] This description of the family includes the word-size, the size and addresses of the CPU and RMEM registers and (an encoding) of the instruction set $\mathcal{I}$.

[9] This latter property will be useful in a real-world deployment, where the computation also modifies program data which might then be returned to the hard drive to be used by another application.

compiles a programming W for RAM $\mathcal{R}$ into a new programming $\widetilde{W}$ for $\mathcal{R}$ by invocation of algorithm $\mathtt{Comp}$; $\mathtt{Ch}$ then emulates an execution of $\widetilde{W}$ on $\mathcal{R}$, i.e., emulates its CPU cycles and stores its entire state at any given point. The adversary is allowed to inject a virus of his choice on any location in the memory $\mathtt{MEM}$. Eventually, the challenger executes the *(virus) detection procedure*: It computes a challenge c by invocation of algorithm $\mathtt{Chal}(z, K_v)$, and then feeds input $(\mathtt{check}, c)$ to the emulated RAM and lets it compute the response y.

To capture worst case attack scenarios, we allow the adversary to inject his virus at any point during the RAM emulation and make no assumption as to how many rounds the RAM executes after the virus has been injected and before it receives the challenge. More concretely, we allow the adversary to specify the number $\rho_{\mathtt{pre}}$ of rounds to be executed before the detection process kicks in, the index $\rho_{\mathtt{att}}$ of the round in which $\mathtt{Adv}$ wants the virus to be injected on the memory, and the number of rounds $\rho_{\mathtt{vd}}$ of the virus detection that the RAM will execute. Furthermore, we make no assumptions on how much information the adversary holds on the original programming W or on the inputs of $\mathcal{R}$, by assuming that $\mathtt{Adv}$ knows W and can decide/see the entire sequence of inputs.

The formal description of the security game $\mathcal{G}_{\mathtt{VDS}}^{\mathcal{R},\mathcal{V},W}$ is given in Figure 1. For simplicity we describe the game for the case where the adversary injects a virus only once, but our treatment can be extended to repeated injections.

Both the adversary $\mathtt{Adv}$ *and the challenger* $\mathtt{Ch}$ *know the RAM* $\mathcal{R} = (\mathcal{C}, \mathtt{MEM})$*, the programming* W*, and the specification of a VDS* $\mathcal{V} = (\mathtt{Gen}, \mathtt{Comp}, \mathtt{Chal}, \mathtt{Resp}, \mathtt{Ver})$.

1. The adversary $\mathtt{Adv}$ chooses (a sequence of) inputs $x = x_1||\ldots||x_i$ for $\mathcal{R}$, along with three numbers $\rho_{\mathtt{pre}} = poly(k), \rho_{\mathtt{att}} = poly(k)$, and $\rho_{\mathtt{vd}} = poly(k)$; additionally, $\mathtt{Adv}$ chooses a virus $\mathbf{v}$ $\mathtt{Adv}$ hands $(x, \rho_{\mathtt{pre}}, \rho_{\mathtt{att}}, \rho_{\mathtt{vd}}, \mathbf{v})$ to the challenger $\mathtt{Ch}$.
2. $\mathtt{Ch}$ chooses $(K_c, K_v) \xleftarrow{\$} \mathtt{Gen}(1^k)$ and computes $\widetilde{W} \xleftarrow{\$} \mathtt{Comp}(\mathcal{R}, W, K_c)$.
3. $\mathtt{Ch}$ internally emulates $\rho_{\mathtt{pre}}$ rounds of an execution of $\mathcal{R}$ on input x and programming $\widetilde{W}$ (if $\mathcal{R}$ halts, $\mathtt{Ch}$ proceeds to Step 4) where it stores in a counter ρ (initially set to $\rho := 1$) the current round index (i.e., at the end of each round, ρ is increased by 1). During this emulation, if ρ reaches $\rho_{\mathtt{att}}$ then at the beginning of round $\rho_{\mathtt{att}}$ $\mathtt{Ch}$ emulates an injection of $\mathbf{v}$ onto the memory $\mathtt{MEM}$.
4. As soon as Step 3 completes (i.e., $\rho_{\mathtt{pre}}$ rounds have been executed or $\mathcal{R}$ halts), $\mathtt{Ch}$ executes the following *virus detection* procedure:
   i. $\mathtt{Ch}$ chooses $z \xleftarrow{\$} Inp_{\mathtt{chal}}$ and computes $c \xleftarrow{\$} \mathtt{Chal}(z, K_v)$.
   ii. $\mathtt{Ch}$ writes $(\mathtt{check}, c)$ on $\mathcal{R}$'s input register, and executes $\rho_{\mathtt{vd}}$ additional rounds of $\mathcal{R}$ (if $\mathcal{R}$ halts then go to Step 3). As in Step 3, if during this execution the round counter ρ reaches $\rho_{\mathtt{att}}$ then at the beginning of round $\rho_{\mathtt{att}}$ $\mathtt{Ch}$ emulates an injection of $\mathbf{v}$ onto the memory $\mathtt{MEM}$.
5. Denote by y the (concatenation of) the strings which $\mathcal{R}$ writes on its output register during Step 2. $\mathtt{Ch}$ computes $b \xleftarrow{\$} \mathtt{Ver}(K_v, z, c, y)$.

Figure 1: The Virus Attack Game $\mathcal{G}_{\mathtt{VDS}}^{\mathcal{R},\mathcal{V},W}$

The security definition for a VDS is provided in the following. In the definition we use the following terminology: we say that a programming W is *$(\mathcal{R},\mathcal{V})$-consistent* for $\mathcal{R}_k$ if the compiled version $\widetilde{W}$ of W fits in the random access memory $\mathtt{MEM}_k$ of $\mathcal{R}_k$ (a formal specification can be found in the full version).

Definition 2. We say that a virus detection scheme $\mathcal{V} = (\mathtt{Gen}, \mathtt{Comp}, \mathtt{Chal}, \mathtt{Resp}, \mathtt{Ver})$ is secure for RAM family $\mathcal{R}$ if it satisfies the following properties:

1. $\mathcal{V}$ is verification correct, compilation correct, and self-responsive.
2. For sufficiently large k for any $(\mathcal{R},\mathcal{V})$-consistent programming W for $\mathcal{R}_k$ and any polynomial adversary $\mathtt{Adv}$ in the game $\mathcal{G}_{\mathtt{VDS}}^{\mathcal{R}_k,\mathcal{V},W}$ who injects a valid non-empty virus:
$$\Pr[b = 1] \leq \mu(k),$$
where μ is some negligible function, and the probability is taken over the random coins of $\mathtt{Adv}$ and $\mathtt{Ch}$.

In our constructions we restrict our statements to certain class of programming that satisfy some desirable properties making the compilation easier, and/or to certain classes of adversaries (e.g., adversaries injecting their virus to consecutive memory locations). We will then say that the corresponding VDS is *secure with respect to the given class of programmings and/or adversaries.*

The Repeated Detection Game. Definition 2 requires that the adversary is caught even when he injects its virus while the RAM is executing the $\mathtt{Resp}$ algorithm. A relaxed security game, which also provides a useful guarantee for practical purposes has the virus detection (challenge/response) procedure executed multiple times periodically (on the same compiled programming); the requirement is that if the adversary injects its virus to the RAM at round ρ, then he will be caught by the first invocation of the virus detection procedure which starts *after round* ρ. Note that all executions use *the same* compiled RAM program and therefore the same key K. The formal security definition is similar to Definition 2 but requires that any virus that is injected in the RAM will be caught *in the first virus detection attempt performed after the injection.*

5. A VDS FOR MODERATE VIRUSES

In this section we provide a VDS which is secure in the continuous-injection model assuming the virus has a constant number of words. This covers a wide class in reality, as viruses are usually several words long. Nonetheless, in Section 6 we show how to get rid of the length and the continuous injection restrictions.

Our construction proceeds in three steps. First step we construct a VDS $\mathcal{V}_1$ which achieves a weaker notion of security that, roughly, does not have self-responsiveness. In a second step, we show how to transform $\mathcal{V}_1$ into a VDS $\mathcal{V}_2$ which is secure (and self-responsive) in the repeated detection game. Finally, in a third step we show how to transform $\mathcal{V}_2$ into a VDS $\mathcal{V}_3$ which is secure in the standard detection game. We point out that the VDSs $\mathcal{V}_1$ and $\mathcal{V}_2$ are not just steps towards building $\mathcal{V}_3$, but can be useful in applications where their corresponding (weaker) security is satisfactory.

A VDS without self-responsiveness We start by describing a VDS $\mathcal{V}_1 = (\mathtt{Gen}_1, \mathtt{Comp}_1, \mathtt{Chal}_1, \mathtt{Resp}_1, \mathtt{Ver}_1)$ which achieves security without self-responsiveness. More concretely, the corresponding attack-game $\mathcal{G}_{\mathtt{VDS}-}^{\mathcal{R},\mathcal{V},W}$ is derived from the standard attack-game $\mathcal{G}_{\mathtt{VDS}}^{\mathcal{R},\mathcal{V},W}$ by modifying the

detection procedure so that in Step 2 of the game $\mathcal{G}_{\mathtt{VDS}}^{\mathcal{R},\mathcal{V},W}$ (Figure 1), instead of emulating $\mathcal{R}$ on the compiled programming $\widetilde{W}$ and input $(\mathtt{check}, c)$ to compute $y = \mathtt{Resp}(c, \widetilde{W})$, the challenger evaluates $y \xleftarrow{\$} \mathtt{Resp}(c, \widetilde{W})$ himself.

At a high level, the idea of our construction is as follows: The key generation algorithm $\mathtt{Gen}_1$ samples a k-bit key K for a symmetric-key cryptosystem, i.e., $K_c = K_v = K \xleftarrow{\$} \{0,1\}^k$. Given key K, the algorithm $\mathtt{Comp}_1$ computes an additive sharing $\langle K \rangle$ of K and fills the entire memory $\mathtt{MEM}$ by interleaving a different share of $\langle K \rangle$ between every two words in the original programming. More precisely, $\mathtt{Comp}$ compiles a programming W for a RAM $\mathcal{R}$ into a new programming $\widetilde{W}$ for $\mathcal{R}$ constructed as follows: Between any two consecutive words w_i and w_{i+1} of W the compiler interleaves a uniformly chosen k-bit string $K^{i,i+1} = K_1^{i,i+1} || \ldots || K_{\frac{k}{L}}^{i,i+1}$, where each $K_j^{i,i+1} \in \{0,1\}^L$. In the last k bits of the compiled programming (i.e., after the last word $w_{|W|-1}$) the string $K^{last} = K \oplus \bigoplus_{i=0}^{|W|-2} K^{i,i+1}$ is written.[10]

To ensure that the compiled programming $\widetilde{W}$ executes the same computation as W we need to ensure that while being executed it "jumps over" the locations where keys are stored (as the keys are only to be used in the detection procedure). For this purpose, we do the following modification: After each word w_j of the original program we put a $(\mathtt{jumpby}, n)$ instruction where n is the number of key-shares between this and the next W-word in the compiled programming. Similarly, we modify any "jump" instructions of the original programming W to point to the correct locations in $\widetilde{W}$. Note that this modification is not necessarily applicable for arbitrary classes of codes, and for certain self-modifying code it might even be infeasible. Our goal here is to demonstrate applicability of our method, therefore, in the following we restrict to the class of programmings W with the following four properties: **(1)** While executing, W accesses only the first $|W|$ $\mathtt{MEM}$ locations. **(2)** The only type of instructions in W which change the program flow (i.e., modify the program counter) are (conditional) "jumps", that skip constantly many words. (These are sufficient for implementing efficiently control-flow commands such as while- and for-loops.) **(3)** During its execution in RAM $\mathcal{R}$, W does not write any (new) instructions to the memory $\mathtt{MEM}$ (but might insert data) and and might only overwrite locations where instructions other than $\mathtt{no\_op}$ are written. **(4)** The only instructions in W that access the random access memory RMEM are $(\mathtt{read}, i, j)$ (which loads the word from $\mathtt{MEM}[i]$ onto the CPU register $\mathtt{REG}[j]$), $(\mathtt{write}, j, i)$ (which stores the word from $\mathtt{REG}[j]$ to the memory logation $\mathtt{MEM}[i]$), and the built-in load command that is executed at the beginning of each CPU cycle and loads the contents of $\mathtt{MEM}[\mathtt{pc}]$ to a special CPU register. All other instructions define operations on values of the CPU registers. Furthermore, we assume that $\mathtt{read}$ is used to only load data and not instructions on the CPU (i.e., an instruction is executed only if it is loaded by the special built-in "load" command).

We refer to a programming W which satisfies the above conditions as a *non-self-modifying structured programming* for $\mathcal{R}$. Without loss of generality, we shall allow throughout this text that any considered RAM has the above instructions, i.e., $\mathtt{read}$, $\mathtt{write}$, $\mathtt{jumpby}$, and $\mathtt{jumpby\_if}$, as part of its CPU instruction set $\mathcal{I}$. Observe, that the above specification is sufficient for any structured program, i.e., a program which uses while- and for-loops but no "goto"s. As a fact, writing structured programs is considered a good practice and most programmers avoid writing self-modifying code as it can be source of bugs. Furthermore, classical results in programming languages imply that any program can be compiled to a structured program with a low complexity overhead [5, 34]. Nonetheless, compilers at times do generate self-modifying code in order to optimize performance. Extending our results to such code is one of many interesting research direction.

In the full version of this work, we describe (and formally prove the correctness of) a process, called $\mathtt{Spread}$, which spreads such programming to allow for enough space between the words to fit the key-shares and adds the extra "jump" instructions to preserve the right program flow. The compiler $\mathtt{Comp}_1$ uses the process $\mathtt{Spread}$ to translate, as sketched above, a non-self-modifying structured programming W for a RAM $\mathcal{R} = (\mathcal{C}, \mathtt{MEM})$ into a programming $\widetilde{W}$ for $\mathcal{R}$. For simplicity, we assume that $|\mathtt{MEM}| = (k/L + 2)|W|$ (if this is not the case we can expand W with $\mathtt{no\_op}$ instructions prior to computing and/or pad the last positions of $\mathtt{MEM}$ with extra key shares).

To complete the description of VDS $\mathcal{V}_1$ we need to describe the remaining three algorithms $\mathtt{Chal}_1, \mathtt{Resp}_1$, and $\mathtt{Ver}_1$. The algorithm $\mathtt{Chal}_1$ chooses a random string $x \in \{0,1\}^k$, and computes the challenge as a one-time pad encryption of x with key K; i.e., $c \leftarrow \mathtt{Chal}(x, K) = \mathtt{Enc}_K(x) = x \oplus K$. The algorithm $\mathtt{Resp}_1$ works as follows: On input the challenge $c = \mathtt{Enc}_K(r)$ and the compiled programming $\widetilde{W}$, it reconstructs K by summing up all its shares as retrieved from $\widetilde{W}$, and outputs a decryption of the challenge under the reconstructed key, i.e., outputs $y = c \oplus K$. The corresponding verification algorithm $\mathtt{Ver}_1$ simply verifies that $y = x$.

The security of $\mathcal{V}_1$ follows from the fact that an adversary who injects a long-enough virus in consecutive memory locations will overwrite a linear number of bits of some key-share which will information-theoretically destroy an analogous amount of decryption-key bits making it infeasible to correctly decrypt the challenge and thus pass the VDS check.

Theorem 1 *Assuming $\mathcal{R}$ is a complete RAM family the VDS $\mathcal{V}_1$ is secure for $\mathcal{R}$ without self-responsiveness with respect to the class of all non-self-modifying structured programmings and the class of adversaries who injects a virus $\mathbf{v}$ with $|\mathbf{v}| \geq 3$ on consecutive memory locations.*

Adding Self-Responsiveness. We next modify $\mathcal{V}_1$ so that it is secure (with self-responsiveness) in the repeated detection game. Towards this direction, prior to applying the above compilation strategy we extend the given sound programming W to also implement the corresponding response algorithm. The completeness of $\mathcal{R}$ and the universality of structured non-self-modifying code [5, 34] ensure that given that $\mathcal{R}$ has sufficient memory, there exists a programming $W_{\mathtt{Resp}}$ which implements $\mathtt{Resp}$ on $\mathcal{R}$. (Note that the algorithm $\mathtt{Resp}_1$ above is deterministic.) Thus we can combine $W_{\mathtt{Resp}}$ and W via a threading mechanism to some new programming $\mathrm{Emb}(W, W_{\mathtt{Resp}})$ which, when executed (periodically) checks if a new $(\mathtt{check}, c)$-input is provided, and if so stores the CPU state on free memory locations and changes the program counter to point to the location where $W_{\mathtt{Resp}}$ is

[10] Wlog, here we implicitly assume that $(|\mathtt{MEM}| - |W|) = 0 \mod k$ so that the keys fit exactly in the memory. The general case can also be easily treated.

stored; once $\texttt{Resp}$ has produced output, it restores the CPU to its prior state and continues the execution of W.[11]

Although it might seem that using the above trick combined with the response algorithm $\texttt{Resp}_1$ from the previous section, i.e., using $\text{Emb}(W, W_{\texttt{Resp}_1})$, takes care of the self-responsiveness issue, this is not the case, i.e., the resulting VDS in not secure. Informally, the reason is that it only detects attacks (virus injections) that occur *outside* the virus detection procedure. In more detail, a possible adversarial strategy is to inject the virus while the RAM is computing the response to a challenge, and in particular as soon as the key K has been reconstructed on the CPU and is about to be used to decrypt the challenge. By attacking at that exact point, the adversary might be able to inject an "intelligent" virus onto $\texttt{MEM}$ while the key is still in the CPU, restore the key back to the memory and use it to pass the current as well as all future virus detection attempts.

To protect against the above attack we use the following technical trick: instead of using a single compilation-key K of length k we use a $2k$-bit key K which is parsed as two k-bit keys $K_{\texttt{od}}$ and $K_{\texttt{ev}}$ via the following transformation: Let $K = x_1 || \dots || x_{\frac{2k}{L}}$, where each x_i is a word.[12] Then $K_{\texttt{od}}$ is a concatenation of the odd-indexed words, i.e., x_i's with $i = 1 \mod 2$, and $K_{\texttt{ev}}$ is a concatenation of the even indexed words. Now the challenge algorithm outputs a double encryption of z with keys $K_{\texttt{od}}$ and $K_{\texttt{ev}}$, i.e., $c = \texttt{Enc}_{K_{\texttt{od}}}(\texttt{Enc}_{K_{\texttt{ev}}}(z))$. In order to decrypt, the response algorithm does the following: First it reconstructs $K_{\texttt{od}}$ by XOR-ing the appropriate shares, and uses it to decrypt c, thus computing $\texttt{Enc}_{K_{\texttt{ev}}}(z)$. Subsequently, it erases $K_{\texttt{od}}$ from the CPU register (e.g, by filling the register where $K_{\texttt{od}}$ is stored with 0's) and *after the erasure completes* it starts reconstructing $K_{\texttt{ev}}$ and uses it (as above) to decrypt $\texttt{Enc}_{K_{\texttt{ev}}}(z)$ and output $y = z$.

Intuitively, the reason why the above idea protects against injections occuring during a detection-procedure execution in the repeated detection game, is that in order to correctly answer the challenge, the virus needs both $K_{\texttt{od}}$ and $K_{\texttt{ev}}$. However, the keys are never simultaneously written in the CPU. Thus if the adversary injects the virus before $K_{\texttt{od}}$ is erased he will overwrite bits from a share of $K_{\texttt{ev}}$ (which at that point exists only in the memory); thus he will not be able to decrypt the challenge. Otherwise, if the adversary injects the virus after $K_{\texttt{od}}$ has been erased from the CPU, he will overwrite bits from a share of $K_{\texttt{od}}$; in this case he will successfully pass this detection attempt, but will fail the next detection attempt.

But we are still not done, because the above argument cannot work with any encryption scheme, e.g., it fails if we instantiate $\texttt{Enc}(\cdot)$ with one-time-pad encryption as in the previous section. To see why, assume that we use $c = z \oplus K_1 \oplus K_2$. Because the input register is read-only (for the RAM) once c is given it cannot be erased. Furthermore, from c and $y = z$ (which is the output of $\texttt{Resp}$) one can recover $K_{\texttt{od}} \oplus K_{\texttt{ev}} := c \oplus y$. Now if the adversary injects its virus after y is computed but while $K_{\texttt{ev}}$ is still on the CPU's register he can easily recover both $K_{\texttt{od}}$ and $K_{\texttt{ev}}$ and answer all future challenges in an accepting manner. Thus we cannot use the one-time pad encryption as in the VDS $\mathcal{V}_1$. In fact, we cannot even use a standard CCA2 secure cipher for $\texttt{Enc}(\cdot)$ as the virus will have access to a big part of the private key and the cyphertext and standard CCA2 encryption does not account for that (recall that we only require the virus to overwrites a portion of a key share).

Instead we make use of a leakage resilient encryption scheme, which is secure as long as the adversary's probability of guessing the key is negligible even when one leaks a big part of the key. A (public-key) encryption scheme satisfying the above property (with CPA security) was suggested by Dodis, Goldwasser, Kalai, Peikert, and Vaikuntanathan [12]. [13] We point out that the compiler $\texttt{Comp}_2$ will only use the compilation key (i..e, the secret key of the scheme in [13] plays the role of the compilation key) and ignore the public key. In the following we assume that all the remainder algorithms of VDS $\mathcal{V}_2$ use such a public-key leakage resilient double encryption scheme. Thus the corresponding key generation algorithm $\texttt{Gen}_2$ generates two pairs of keys for such a scheme, where the secret (compilation) keys are used by $\texttt{Comp}_2$, whereas all other algorithms in $\mathcal{V}_2$ use the corresponding verification (public) keys.

There are still a couple of details that need to be taken care of in the design of the appropriate response algorithm $\texttt{Resp}_2$ used in $\text{Emb}(W, W_{\texttt{Resp}_2})$. First it should never copy key-shares on other memory locations (this will ensure that the key-shares which the virus overwrites are not recoverable). Second, we need to make sure that $\text{Emb}(W, W_{\texttt{Resp}_2})$ is still able to access key locations, even after it has been compiled. (Recall that the compiler re-arranges the commands $\texttt{read}$ and $\texttt{write}$ to avoid these locations). We can resolve this by a simple technical trick: We make $W_{\texttt{Resp}_2}$ use a special read command $(\texttt{read\_key}, i, j, \ell)$ for accessing the key shares, which reads the i-th word of the jth key-share and stores it on register $\texttt{REG}[\ell]$. According to the rules for spreading out the programming from the previous section, the compiler will not modify the address accessed by this command when $\text{Emb}(W, W_{\texttt{Resp}_2})$ is compiled as it is not a $\texttt{read}$ or $\texttt{write}$ instruction. This extra $(\texttt{read\_key}, i, j, \ell)$ instruction can be easily implemented in machine code by a combination of $\texttt{read}$ and $\texttt{jump}$ commands.

Having described $\texttt{Resp}_2$ we can now provide the detailed description of the corresponding compiler $\texttt{Comp}_2$. The compiler uses the programming $\text{Emb}(W, W_{\texttt{Resp}_2})$ described above, and the compiler $\texttt{Comp}_1$ from the previous section but using the two compilation (secret) keys in tandem. Denote by $\mathcal{V}_2$ the VDS $(\texttt{Gen}_2, \texttt{Comp}_2, \texttt{Chal}_2, \texttt{Resp}_2, \texttt{Ver}_2)$, where $\texttt{Gen}_2$, $\texttt{Comp}_2$, $\texttt{Chal}_2$, and $\texttt{Resp}_2$ are as described above, and $\texttt{Ver}_2$ is identical to $\texttt{Ver}_1$. As in Theorem 1, we assume wlog that the RAM family has sufficient memory to accommodate $\widetilde{W} = \texttt{Comp}_2(\text{Emb}(W, W_{\texttt{Resp}_2}))$.

Theorem 2 *Assuming $\mathcal{R}$ is a complete RAM, if the encryption scheme used in $\mathcal{V}_2$ is CPA secure against an adversary who learns all but $\ell = \omega(\log k)$ bits of the secret key [12], then the VDS $\mathcal{V}_2$ is secure in the repeated detection model with respect to the class of non-self-modifying structured programmings and the class of adversaries who injects a virus $\mathbf{v}$ with $|\mathbf{v}| \geq 4$ on consecutive memory locations.*

Obtaining Standard Security. The final step is to com-

[11] Modern architectures use the program stack for efficiently implementing such a multi-thread computation; in the full version we describe a simple low level implementation in our RAM model.

[12] Wlog we assume that $k = Lq$ for some q.

[13] For technical reasons we cannot use the (CCA) symmetric encryption with auxiliary input [13]. For details we refer to the proof of Theorem 2 in the full version.

pile the VDS $\mathcal{V}_2$ into a VDS $\mathcal{V}_3$ which is secure in the standard model. The transformation is to a large extend generic and can be applied to most self-responsive VDSs as long as the algorithm $\texttt{Resp}$ can be described as an *invertible* RAM program, i.e., a program which at the end of its execution leaves the memory in the same state it was at the beginning. Given sufficiently memory, that this is the case for the algorithms described here, as they only need to compute exclusive-ORs of the key shares and then use it to decrypt the challenge. Modern CPUs can even perform these operations without ever changing the contents of the memory.

Our transformation also assumes a hash function $h(\cdot)$ which can be efficiently computed by an invertible programming on the RAM $\mathcal{R}$. Most known hash functions can be implemented as such by writing only on CPU registers and/or on temporary memory locations (swap memory) which can be erased at the end of their invocation. We convert the VDS $\mathcal{V}_2 = (\texttt{Gen}_2, \texttt{Comp}_2, \texttt{Chal}_2, \texttt{Resp}_2, \texttt{Ver}_2)$ which is secure in the repeated detection model into a VDS $\mathcal{V}_3 = (\texttt{Gen}_3, \texttt{Comp}_3, \texttt{Chal}_3, \texttt{Resp}_3, \texttt{Ver}_3)$ secure in the single detection model as follows:

1. The algorithm $\texttt{Gen}_3$ is the same as $\texttt{Gen}_2$.
2. The algorithm $\texttt{Resp}_3$ works as follows on input a challenge c and a programming $\widetilde{W}$: **(i)** It executes the invertible programming for $h(\cdot)$ which computes a complete hash $y_h = h(\widetilde{W}||c)$ of the contents of memory RMEM (at the beginning of the evaluation of $h(\cdot)$) concatenated with the challenge ciphertext,[14] outputs y_h to the user, and then erases all memory locations and CPU registers that it wrote on. **(ii)** It executes the invertible programming for $\texttt{Resp}_2$ and outputs its output $y = \texttt{Resp}_2(c, \widetilde{W})$. **(iii)** It executes again the invertible programming for $h(\cdot)$ computes again a complete hash $y'_h = h(\widetilde{W}'||c)$ where $\widetilde{W}'$ denote the current memory RMEM and outputs y'_h.
3. The algorithms $\texttt{Comp}_3$ is the same as $\texttt{Comp}_2$ but uses $\texttt{Resp}_3$ instead of $\texttt{Resp}_2$, i.e., first generates the programming $\text{Emb}(W, W_{\texttt{Resp}_3})$, and then spreads it and interleaves the key-shares.
4. $\texttt{Chal}_3$ is the same as $\texttt{Chal}_2$.
5. $\texttt{Ver}_3$ is modified as follows: Let K_v and z denote the inputs of $\texttt{Chal}_3$ in the attack game $\mathcal{G}^{\mathcal{R},\mathcal{V},W}_{\texttt{VDS}}$ (Figure 1), and let y_h, y, and y'_h denote the outputs of the detection procedure. Then $\texttt{Ver}_3(\cdot, z, \cdot, (y_h, y, y'_h))$ outputs 1 if $y = z$ and $y_h = y'_h$, otherwise it outputs 0.

The above modification takes care of adversaries that submit a virus to be injected during the computation of $\texttt{Resp}_3$: If the injections occurs before the first evaluation of the hash function has been erased, then the security of $\mathcal{V}_2$ in the repeated detection model ensures that the adversary will be caught with overwhelming probability. Otherwise, if the virus is injected after y_h has been erased, then in order to produce a valid reply, the virus will need to compute y_h. Because by assumption, the adversary overwrites at least $L = \mathcal{O}(k)$ bits of a key share which he cannot recover, the security of $h(\cdot)$ will guarantee that the probability that the virus outputs y_h is negligible in k.

Theorem 3 *Assuming $\mathcal{R}$ is a complete RAM, if the encryption scheme used in $\mathcal{V}_2$ is CPA secure against an adversary who learns all but $\ell = \omega(\log k)$ bits of the secret key [12], then the VDS $\mathcal{V}_3$ is secure with respect to the class of non-self-modifying structured programmings and the class of adversaries who injects a virus $\mathbf{v}$ with $|\mathbf{v}| \geq 4$ on consecutive memory locations.*

[14]This will ensure that, even if one uses this scheme for repeated detections, the hash will be different in each invocation of the detection process.

6. VDS FOR ALL VIRUSES

The constructions from the previous section are secure only against (sufficiently) long viruses (i.e., more than two words). In this section we describe a VDS $\mathcal{V}_4$ which achieves security $2^{-\mathcal{O}(k)}$ for any desirable value of the security parameter k, independent of the virus length, and in particular even when the virus affects only part of a single word (i.e., is a few bits long). Injection of such a virus can be easily captured by assuming a bit-by-bit injection (see full version for details).

The basic difficulty in designing a VDS which tolerates a virus of arbitrary small size is that depending on the actual programming we compile, an "intelligent" short virus that is injected on the position of the first instruction executed might cause the RAM to jump to a location that allows the virus to take over the entire computation (e.g., by *return-oriented programming* [40]). To prevent such an attack and ensure that even such viruses will be caught we use the following idea: We use a compiler similar to $\texttt{Comp}_3$, but we include, for each program word and pair of key-shares, message authentication codes (MACs) which we check every time we load a program word on the CPU.

A MAC consists of a pair of algorithms $(\texttt{Mac}, \texttt{Ver})$, where $\texttt{Mac} : \mathcal{M} \times \mathcal{K} \to \mathcal{T}$ is the authentication algorithm (i.e., for a message $m \in \mathcal{M}$ and a key $\texttt{vk} \in \mathcal{K}$, $t = \texttt{Mac}(m, \texttt{vk}) \in \mathcal{T}$ is the corresponding authentication tag), and $\texttt{Ver} : \mathcal{M} \times \mathcal{T} \times \mathcal{K} \to \{0,1\}$ is the verification algorithm (i.e., for a message $m \in \mathcal{M}$, an authentication tag $t \in \mathcal{T}$ and a key $\texttt{vk} \in \mathcal{K}$ $\texttt{Ver}(m, t, \texttt{vk}) = 1$ if and only if t is a valid authentication tag for m with key $\texttt{vk}$). We let $\mathcal{M}$ include all strings of length at most k, $\mathcal{K} = \{0,1\}^{2k}$, and $\mathcal{T} = \{0,1\}^k$. Such a MAC can be constructed as follows: Let $GF(2^k)$ be the field of characteristic two and order 2^k. (Every $x \in GF(2^k)$ can be represented as a k-bit string and vice-versa). Let also $\texttt{vk} = \texttt{vk}_1||\texttt{vk}_2 \in \{0,1\}^{2k}$ where $\texttt{vk}_i \in \{0,1\}^k$ for $i \in \{1,2\}$. Then $\texttt{Mac}(m, \texttt{vk}) = \texttt{vk}_1 \cdot m + \texttt{vk}_2$, where $+$ and $\cdot$ denote the field addition and multiplication in $GF(2^k)$, respectively.[15] (If $|m| < k$ then pad it with appropriately many zeros.) The verification algorithm simply checks that the message, tag, and key satisfy the above condition. An important property of the above MAC is that for a given key $\texttt{vk}$, every $m \in \mathcal{M}$ has a *unique* authentication tag that passes the verification test. Furthermore, the probability of any (even computationally unbounded) adversary learning a MAC tag to guess the corresponding key is at most 2^{-k}.

Similar to $\texttt{Comp}_3$, our new compiler $\texttt{Comp}_4$ compiles the programming $\text{Emb}(W, W_{\texttt{Resp}_4})$, i.e., W combined with the response algorithm $\texttt{Resp}_4$ described below, into a new programming $\widetilde{W}$ which interleaves additive shares $K^{i,i+1}_{\texttt{od}}$ and $K^{i,i+1}_{\texttt{ev}}$ of decryption keys $K_{\texttt{od}}$ and $K_{\texttt{ev}}$ between any two program words w_i and w_{i+1} and adds the appropriate jump command to ensure correct program flow. The difference is that $\texttt{Comp}_4$ also adds MAC tags $t^i_{\texttt{od}} = \texttt{Mac}(\widetilde{w}_i||\widetilde{w}_{i+1}, K^{i,i+1}_{\texttt{od}})$

[15]Most modern CPUs are equipped with an arithmetic logic unit the can perform modular arithmetic operations.

and $t^i_{\mathtt{ev}} = \mathtt{Mac}(\widetilde{w}_i||\widetilde{w}_{i+1}, K^{i,i+1}_{\mathtt{od}})$.[16] To visualize it: $\mathtt{Comp}_4$ expands every word $w_i \in \mathrm{Emb}(W, W_{\mathtt{Resp}_4})$, as $\widetilde{w}_i||(\mathtt{jumpby}, n)||K^{i,i+1}_{\mathtt{od}}||K^{i,i+1}_{\mathtt{ev}}||\mathtt{Mac}(\widetilde{w}_i||(\mathtt{jumpby}, n), K^{i,i+1}_{\mathtt{od}})||\mathtt{Mac}(\widetilde{w}_i||(\mathtt{jumpby}, n), K^{i,i+1}_{\mathtt{ev}})$.

As we prove, this ensures that any virus, no matter how small, which is written consecutively in MEM in the bit-by-bit injection setting, will either have to overwrite a long number of bits of some key K_i, or will create an inconsistency in at least one of the MAC's with overwhelming probability. To exploit the power of the MACs we need to ensure that during the program execution, before loading any word the CPU first verifies its MACs. Therefore, we assume that, by default, the CPU loads values from the memory RMEM to its registers via a special load instructions $(\mathtt{read\_auth}, i, j)$, which, first, verifies both MACs of the word in location i of the memory with the corresponding keys, and only if the MACs verify, it keeps the word on the CPU. If the MAC verification fails, then $(\mathtt{read\_auth}, i, j)$ deletes at least one of the key-shares from the memory, thus introducing an inconsistency that will be caught by the detection procedure.

Note that $(\mathtt{read\_auth}, i, j)$ makes no tamper-resiliency assumption on the memory or the CPU. It only needs for the architecture to provide us with this simple piece of microcode, and by engineering the CPU's hardware in a smart way (so that these operations are done on the circuit level) we expect to be able to reduce the effect of our compiler to a barely noticeable slowdown. Fortunately, the new architecture that Intel recently announced promises to embed such microcode in its next-generation microchips [35]. Luckily, the new architecture that Intel recently announced promises to allow for support of such a microcode in its next generation microchips [35]. The implementation details on the newly designed architecture are the subject of future work.

Observe that the original programming W might include $(\mathtt{read}, i, j)$ instructions which we will replace by $(\mathtt{read\_auth}, i, j)$. Furthermore, to ensure that the compiled program will not introduce inconsistencies, we replace every $(\mathtt{write}, j, i)$ instruction of the programming with microcode write_auth which, when writing a word in the memory it also updates the corresponding MAC tags. Unlike read_auth, this write_auth does *not* require any change in the architecture or additional assumptions as it can be implemented in assembly code which can be included in W. Nonetheless, a smart redesign of the CPU circuit (e.g., extensions of its ALU) to implement write_auth in hardware could give a speed up that would make its effect nearly noticeable.

A formal description of the compiler $\mathtt{Comp}_4$ is derived along the lines of $\mathtt{Comp}_3$ with the above modifications. We next turn to other three algorithms of the VDS $\mathcal{V}_4$, namely $\mathtt{Gen}_4$, $\mathtt{Chal}_4$, $\mathtt{Resp}_4$ and $\mathtt{Ver}_4$. The key generation $\mathtt{Gen}_4$ is identical to $\mathtt{Gen}_3$ but generates $2k$-bit compilation keys. The algorithms $\mathtt{Chal}_4$ and $\mathtt{Ver}_4$ are identical to the corresponding algorithms from the previous section, where the verification keys are as generated by $\mathtt{Gen}_4$. $\mathtt{Resp}_4$ works as $\mathtt{Resp}_3$ with the following difference: After the first evaluation of the hash function, and before doing anything else, $\mathtt{Resp}_4$ scans the entire memory to ensure that all MAC tags are correct. After this check, $\mathtt{Resp}_4$ continues as $\mathtt{Resp}_3$ would. Observe that for all $(\mathtt{read}, \cdot, \cdot)$ instructions in $W_{\mathtt{Resp}_3}$, the compiler $\mathtt{Comp}_4$ will replace then with corresponding $(\mathtt{read\_auth}, \cdot, \cdot)$ instructions. However, it will not touch the $(\mathtt{read\_key}, i, j, \ell)$ instructions used for reading the keys. Therefore, the keys need not be authenticated. In the following we denote by $\mathcal{V}_4$ the VDS $(\mathtt{Gen}_4, \mathtt{Comp}_4, \mathtt{Chal}_4, \mathtt{Resp}_4, \mathtt{Ver}_4)$.

Theorem 4 *Let $\mathcal{R}$ be a complete RAM. If the encryption scheme used in $\mathtt{Comp}_2$ is CPA secure even against an adversary who learns any $\omega(\log k)$ bits of the secret key, then the VDS $\mathcal{V}_4$ is secure for $\mathcal{R}$ with respect to the class of all non-self-modifying structured programmings and adversaries in the bit-by-bit continuous injection model.*

Removing the Continuous-Injection Assumption. The above VDS detects injections of arbitrary small size, but assuming continuous injection. We now show how to remove this assumptions, at the cost however of an additional modification on the CPU (microcode) specification. (As before, the modification is simple enough to be implemented in hardware which will impose a very small slowdown.) More concretely, as for $\mathcal{V}_4$, we use MACs for authenticating the words. However, to ensure that the adversary cannot forge the MAC by a smart manipulation of the word, keys, and tags, we do not use the actual key-shares $K^{i,i+1}_{\mathtt{od}}$ and $K^{i,i+1}_{\mathtt{ev}}$ in the computation and verification of the above MAC but rather their hash-value, i.e., $H(K^{i,i+1}_{\mathtt{od}})$ and $H(K^{i,i+1}_{\mathtt{ev}})$ for some hash function $H(\cdot)$. [17] Intuitively, this will ensure that if the malware overwrites less than d bits of some of the key-shares it will be unable to guess an appropriate manipulation of the MAC key. If, on the other hand, the adversary overwrites more than d bits of any key-share, then with overwhelming probability he will destroy a super-logarithmic portion of the decryption key and will be, therefore, unable to answer decryption challenges. More concretely, the compiler $\mathtt{Comp}_5$ works as $\mathtt{Comp}_4$ but maps a word w_i of the original programming onto $\widetilde{w}_i||(\mathtt{jumpby}, n)||K^{i,i+1}_{\mathtt{od}}||K^{i,i+1}_{\mathtt{ev}}||\mathtt{Mac}(w_i||(\mathtt{jumpby}, n), H(K^{i,i+1}_{\mathtt{od}}), H(K^{i,i+1}_{\mathtt{ev}}))$. The instructions read_auth and write_auth are modified accordingly to load and read the above combination of word, keys, and MAC as a single chunk and apply the modified MAC algorithm (which uses key-share hashes instead of the key-shares themselves). The remainder algorithms, i.e., $\mathtt{Chal}_5$, $\mathtt{Resp}_5$, and $\mathtt{Ver}_5$ are identical to $\mathtt{Chal}_4$, $\mathtt{Resp}_4$, and $\mathtt{Ver}_4$, respectively.

In the following we denote by $\mathcal{V}_5$ the VDS $(\mathtt{Gen}_5, \mathtt{Comp}_5, \mathtt{Chal}_5, \mathtt{Resp}_5, \mathtt{Ver}_5)$. The security proof of $\mathcal{V}_5$ follows the intuition sketched above and is stated in the following theorem.

Theorem 5 *Let $\mathcal{R}$ be a complete RAM. If the encryption scheme used in $\mathcal{V}_5$ is CPA secure even against an adversary who learns all but $\omega(\log k)$ bits of the secret key, then the VDS $\mathcal{V}_5$ is secure for $\mathcal{R}$ with respect to the class of all non-self-modifying structured programmings and adversaries in the bit-by-bit (non-continuous) injection model.*

7. REFERENCES

[1] T. AbuHmed, N. Nyamaa, and D. Nyang. Software-based remote code attestation in wireless sensor network. In *GLOBECOM'09*, pp. 4680–4687, 2009.

[16] Wlog assume that $L \leq 2k$.

[17] As usually, we will assume that $H(\cdot)$ behaves as a random function, which is very natural to assume since we do not need any compression and we only evaluate it on randomly distributed shares.

[2] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *SP '97*, pp. 65–, 1997.

[3] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *ATEC '00*, pp. 21–21, 2000.

[4] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a board range of memory error exploits. In *SSYM'03*, pp. 8, 2003.

[5] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, May 1966.

[6] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In *EUROCRYPT'97*, volume 1233 of *LNCS*, pp. 37–51, 1997.

[7] B. Chen and R. Morris. Certifying program execution with secure processors. In *HOTOS'03*, pp. 23–23, 2003.

[8] F. B. Cohen. Operating system protection through program evolution. *Comput. Secur.*, 12(6):565–584, October 1993.

[9] 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 *SSYM'98*, pp. 5–5, 1998.

[10] S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Booby trapping software. In *NSPW'13*, pp. 95–106, 2013.

[11] D. Dachman-Soled, F.-H. Liu, E. Shi, and H.-S. Zhou. Locally Decodable and Updatable Non-malleable Codes and Their Applications. In *TCC 2015*, volume 9014 of *LNCS*, pp. 427–450, 2015.

[12] Y. Dodis, S. Goldwasser, Y. T. Kalai, C. Peikert, and V. Vaikuntanathan. Public-key encryption schemes with auxiliary inputs. In *TCC 2010*, volume 5978 of *LNCS*, pp. 361–381, 2010.

[13] Y. Dodis, Y. T. Kalai, and S. Lovett. On cryptography with auxiliary input. In *STOC'09*, pp. 621–630, 2009.

[14] S. Dziembowski, K. Pietrzak, and Daniel Wichs Non-Malleable Codes. In *ICS 2010*, pp. 434–452, 2010.

[15] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS 2012*, 2012.

[16] C. C. Elgot and A. Robinson. Random-access stored-program machines, an approach to programming languages. *J. ACM*, 11(4):365–399, 1964.

[17] S. Faust, P. Mukherjee, J. B. Nielsen, and D. Venturi. A tamper and leakage resilient von neumann architecture. In *PKC 2015*, volume 9020 of *LNCS*, pp. 579–603, 2015.

[18] H. H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *SP'04*, pp. 194, 2004.

[19] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *SP '03*, pp. 62–, 2003.

[20] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *SP '96*, pp. 120–, 1996.

[21] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A minimalist approach to remote attestation. In *DATE '14*, pp. 244:1–244:6, 3001 2014.

[22] M. Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *NSPW '10*, pp. 7–16, 2010.

[23] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *FOCS'13*, pp. 40–49, 2013.

[24] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *SSYM'12*, pp. 40–40, 2012.

[25] O. Goldreich. *Foundations of Cryptography: Basic Tools*, volume 1. Cambridge University Press, Cambridge, UK, 2001.

[26] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, 1998.

[27] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *VEE '06*, pp. 2–12, 2006.

[28] M. Jakobsson and K.-A. Johansson. Practical and secure Software-Based attestation. In *LightSec*, 2011.

[29] M. Jakobsson and G. Stewart. Mobile malware: Why the traditional AV paradigm is doomed, and how to use physics to detect undesirable routines. In *BlackHat*, 2013.

[30] A. Juels and B. S. Kaliski Jr. Pors: proofs of retrievability for large files. In *ACM CCS 07*, pp. 584–597, 2007.

[31] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *SSYM'03*, pp. 21–21, 2003.

[32] E. Klarreich. Perfecting the art of sensible nonsense. *Quanta Magazine*, January 2014.

[33] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for timing-based attestation. In *SP'12*, pp.r 21-23 2012.

[34] R. C. Linger, H. D. Mills, and B. I. Witt. *Structured programming - theory and practice*. The systems programming series. Addison-Wesley, 1979.

[35] 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 *HASP '13*, pp. 10:1–10:1, 2013.

[36] W. S. McPhee. Operating systems integrity in os/vs2. *IBM Systems Journal, 13 Issue 3*, pp. 230–252, 1974.

[37] A. Sahai and B. Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *STOC'14*, pp. 475–484, 2014.

[38] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. Scuba: Secure code update by attestation in sensor networks. In *WiSe '06*, pp. 85–94, 2006.

[39] 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 *SOSP '05*, pp. 1–16, 2005.

[40] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM CCS '07*, pp. 552–561, 2007.

[41] A. Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, 1979.

[42] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *SP'09*, 7(1):26–33, 2009.