CoRReCt: Compute, Record, Replay, Compare to Secure Computations on Untrusted Systems

Felix Dörre[1], Marco Liebel, Jeremias Mechler[1], and Jörn Müller-Quade[1]

KASTEL, Karlsruhe Institute of Technology, Karlsruhe
{felix.doerre,jeremias.mechler,joern.mueller-quade}@kit.edu

Abstract. If the system of an honest user is corrupted, all of its security may be lost: The system may perform computations using different inputs, report different outputs or perform a different computation altogether, including the leakage of secrets to an adversary.

In this paper, we present an approach that complements arbitrary computations to protect against the consequences of malicious systems. To this end, we adapt a well-known technique traditionally used to increase fault tolerance, namely redundant executions on different machines that are combined by a majority vote on the results. However, using this conceptually very simple technique for general computations is surprisingly difficult due to *non-determinism* on the hardware and software level that may cause the executions to deviate.

The *CoRReCt* approach, short for *Compute, Record, Replay, Compare*, considers two synchronized executions on different machines. Only if both executions lead to the same result, this result is returned. Our realization uses virtual machines (VMs): On one VM, the software is executed and non-deterministic events are recorded. On a second VM, the software is executed in lockstep and non-deterministic events are replayed. The outputs of both VMs, which are hosted on different machines, are compared by a dedicated trusted entity and only allowed if they match. The following security guarantees can be proven:

- *Integrity:* If at most one host is corrupted, then the computation is performed using the correct inputs and returns either the correct result or no result at all.
- *Privacy:* If timing side-channels are not considered and at most one host is corrupted, the additional leakage introduced by our approach can be bounded by $\log_2(n)$ bits, where n is the number of messages sent. If timing side-channels are considered and the recording system is honest, the same leakage bound can be obtained.

As VMs can be run on completely different host platforms, *e.g.* Windows on Intel x86-64 or OpenBSD on ARM, the assumption of at least one system being honest is very plausible.

To prove our security guarantees, we provide a proof within a formal model. To demonstrate the viability of our approach, we provide a ready-to-use implementation that allows the execution of arbitrary (networked) x86-64 Linux programs and discuss different real-world applications.

Keywords: Record-Replay · Secure Computation · Non-Determinism.

1 Introduction

When using untrusted systems, even basic properties such as correctness are not guaranteed: A malicious system may change the execution in an arbitrary way, possibly using different inputs or returning wrong results. Depending on the performed task, the consequences of a deviation may be grave, including a total loss of privacy.

Apart from such crass deviations in an execution, the influence of a malicious system may be much more subtle, but the consequences may be just as severe: In today's systems, privacy is often achieved by relying on cryptographic building blocks, for example for encryption or key exchange. Often, the security of these building blocks relies on a) high-quality randomness and b) the secrecy of cryptographic keys. On an untrusted system, neither may be guaranteed. For example, randomness provided by the system may be of low quality, *e.g.* with low entropy, leading to a total loss of privacy. Also, a malicious system could leak sensitive information via side-channels, for example via the timing of network packets or the choice of sequence numbers[1].

Interestingly, even advanced and expensive techniques such as secure multi-party computation [1, 2] may not offer protection in this setting: Many established security notions just distinguish between honest and corrupted parties and only provide security guarantees for parties that are honest. If an honest party (or its system) gets corrupted, no security guarantees are given anymore. This is not only a definitional artifact, but a consequence of the system participating in a secure computation typically holding the input and being responsible for returning the output.

Trusted Computing. Towards protecting against corrupted systems, a popular technique that continues to see widespread use is so-called "trusted computing". By embedding specific modules that act as "roots of trust" into the system and/or its individual components, the goal is to a) provide the ability to verify that the execution has not been tampered with and b) possibly also reduce the attack surface, for example in the case of secure enclaves. For example, trusted computing may allow to verify that the executed software is the intended one (or preventing a system from running "unwanted" software completely) before giving sensitive input. Practical implementations of this system use a Trusted Platform Module (TPM) [3], or focus on securing part of the execution like Intel Software Guard Extensions (SGX) [4] for specific programs and AMD Secure Encrypted Virtualization (AMD SEV) for individual virtual machines. However, the hardware and software components that enable trusted computing in the first place (including the potentially large *trusted computing base*) are usually very complex, often closed-source and proprietary, making their honesty and correctness a very strong assumption that may not be justified in practice. Indeed, if these components are malicious or faulty, for example because the manufacturer is forced by a government agency to embed a backdoor, no security may be provided at all.

[1] `https://github.com/Kicksecure/tirdad`

A Different Approach. This work explores an orthogonal approach for securing a computation, namely by *distributing* the trust between two machines instead of *focusing* it on a single one. By setting up a computation with CoRReCt, only one out of two general-purpose systems needs to be trusted. These systems may come from different manufacturers and run different operating systems. This is in contrast to recent trusted computing techniques such as AMD SEV, Intel SGX or Intel TDX, which are available from one (processor) manufacturer only and currently may have limited operating system support. To this end, we adapt a well-known approach used to increase fault tolerance, namely the parallel execution on multiple heterogeneous systems, whose results are compared (or have a majority vote performed) by a simple application-agnostic comparator. In such a setting, a (not necessarily strict) majority of correct or honest systems leads to a correct output or no output at all, but never to a wrong output.

A challenge in this setting is *non-determinism* that may occur in the execution of complex systems, for example due to timing. In the presence of such non-determinism, performing the same computation on multiple (and even honest) systems may unintentionally lead to different results. To deal with this problem, an established technique is to *record* non-deterministic events in one execution and to *replay* them in the other execution(s). This requires either very similar systems [5] or an appropriate level of abstraction.

However, when considering security and, in particular privacy, this naïve approach may be insufficient: For many tasks, high-quality randomness is necessary. However, randomness is often obtained by harvesting non-deterministic events[2]. In a setting where non-determinism may be controlled (to some extent) by an adversary, the quality of the randomness is no longer guaranteed, even if the replaying system is honest. At the same time, using independent randomness on each system is impossible, as this might lead to inconsistent executions with different results.

In our approach, which we call CoRReCt, short for *compute*, *record*, *replay*, *compare*, we deal with this problem by obtaining cryptographically secure randomness via a special protocol in a precomputation phase. This randomness can then be used for the subsequent computation, greatly reducing an adversary's influence from exploiting non-deterministic behavior. In case of the Linux kernel, we make sure that only secure randomness is used, as added non-determinism can reduce entropy from the point of view of an adversary. For our implementation, we modified the Linux kernel to only use randomness created through a cryptographic protocol.

We demonstrate the usefulness of our approach with the use-cases of i) secure multi-party computations, ii) secure messaging using *Tinfoil chat* [6] and iii) software building.

[2] `https://github.com/torvalds/linux/blob/994d5c58e50e91bb02c7be4a91d5186292a895c8/drivers/char/random.c#L1127`

1.1 Related Work

Redundancy. Duplicating components for fault tolerance is a general approach to increase the reliability against accidental failure. This practice is also applied for general computations, for example in ARM Cortex-R realtime cores [7].

Byzantine Fault Tolerance. A related problem, considered by *byzantine fault tolerance* (BFT) [8, 9], is to achieve consensus between multiple systems where a subset of the systems may arbitrarily misbehave. In particular, this means corrupted systems might send differing information to honest systems. For performance reasons, CoRReCt does not intend to achieve consensus, but merely to *detect* byzantine faults in an execution with two machines. This allows CoRReCt to establish guarantees for the case that no faults occur, but the execution is still influenced by an adversary through allowed behavior.

Record-Replay. Record-replay systems have been investigated for some time, with debugging of non-deterministic applications as their main focus [10, 11]. Recording non-determinism and thereby enabling deterministic replay allows for effective debugging. It provides features like *reverse stepping* to help understand complex bugs, especially those of non-deterministic nature.

The use of recording and replaying non-deterministic behavior of VMs for fault tolerance in case of hardware failure has been explored for a longer time, with early research implementations [12] and newer, practical implementations [5, 13, 14], some of which are production-ready today. Although replaying to enable fault tolerance seems quite similar to our goal, the main objective is to continue execution of a virtual machine (VM) in case one VM host goes offline. That means that in practice, only one of the machines will produce outputs, until a fail-over is decided and the replaying VM takes over. As a consequence, replaying for fault tolerance is usually done on similar hardware, so VMs remain in sync, while still being able to use hardware support for virtualization. As outputs are not compared, these previous approaches do not protect against malicious output produced by a corrupted system. In contrast, our approach relies on multiple hosts, possibly from different vendors, cross-checking the computation results to prevent one host from manipulating them.

The canonical alternative approach for handling non-deterministic behavior, is to remove it. That might be difficult in all cases, but there is work[15], that removes the majority of non-deterministic behavior.

Robust Combiners. Robust combiners allow to combine different implementations of (mainly cryptographic) primitives such that even if an unknown subset (of bounded size) of the implementations is corrupted, the combination is still secure. Such robust combiners not only aim to mitigate accidental failures, but can even deal with completely adversarial implementations. Previous work exists on finding a formal definition and gives various easy combiners like cascade or copying, giving an overview of easy constructions [16]. It has been shown that some robust combiners are impossible to create in a black-box way, *i.e.* by using the different implementations only via their input-output interface and not via their code, for example for oblivious transfer [17, 18].

Robust combiners have also been constructed for more complex hardware components like firewalls [19]. Similar to our approach, a trusted interface that sends the inputs to the individual implementations and performs a check on the outputs (for example a comparison or a majority vote), is used.

1.2 Contribution and Outline

In this work, we propose, model and implement an approach for increasing the security of software execution on untrusted general-purpose systems. In particular, we consider the setting of two untrusted systems, possibly from different vendors and running different operating systems, that execute QEMU Linux VMs in a record-replay manner. To distribute input to these systems as well as to compare their outputs, we use a trusted input-output interface (see Fig. 1). We prove that if one host correctly executes the VM and if the input-output interface is trusted, then the only adversarial influence to the execution constitutes of aborting and controlling non-deterministic behavior. In particular, either the output is correct (relative to the allowed adversarial influence) or no output is given at all.

Our approach is, to the best of our knowledge, the first to take special care to ensure that the used randomness is cryptographically secure, making it suitable for a setting where security and privacy are important. We demonstrate the practicality by realizing a system based on QEMU and measuring its performance.

In more detail, we provide a description of our approach, a threat model as well as a formal model and proof of security in Section 2, discuss possible applications in Section 3 and give an overview of our implementation as well as benchmark results in Section 4.

We stress our goal is *not* to protect the software *inside* the VM from being bug-free, and thereby from malware attacks or hacks. Instead, our focus is ensuring its correct execution and implied security and privacy guarantees for appropriate programs when at least one VM host and the input-output interface are honest.

Our approach is suitable for a setting where a very high level of security is needed, at the expense of performance or system complexity. In particular, CoRReCt may be helpful when trusted general purpose systems cannot be easily obtained or when hardware trojans or supply chain attacks are a valid threat.

2 The CoRReCt Approach

We start the presentation of CoRReCt with an informal description, continue with a threat model and finish with a formal model. Within this model, we state an ideal functionality that captures the security guarantees provided by CoRReCt. Subsequently, we cast our approach as a protocol within our model and prove that it realizes the ideal functionality.

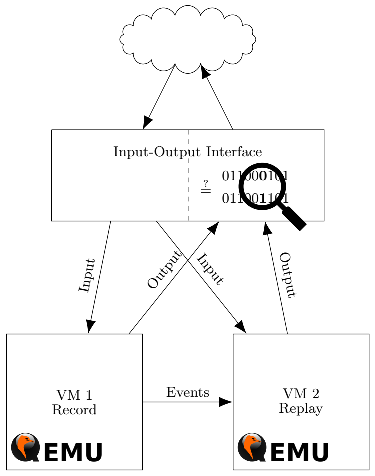

Fig. 1. Architecture of a system protecting a computation using record-replay on x86-64 machine level. Events are QEMU-internal events, inputs and outputs are ethernet packages

2.1 Description

Following the setting illustrated in Fig. 1, we consider an execution with three machines. One machine, called IO, is tasked with distributing inputs to two other machines denoted by P_1 and P_2 that execute the same program. P_1 is called the *recorder*. It executes a Turing machine μ on inputs received by IO and records all non-deterministic events. Conversely, P_2 is called the *replayer*. It receives inputs from IO, as well as a trace of recorded non-deterministic events for each activation from P_1. Using the input and the trace, P_2 also executes μ. Both P_1 and P_2 provide output to IO. To determine the contents of μ's random tape, P_1 and P_2 perform an appropriate protocol. If the outputs of P_1 and P_2 match, IO allows the output. Otherwise, it halts without accepting further inputs or outputs. We assume that P_1, P_2 and IO may directly communicate with each other, without any adversarial influence beyond the corruption of parties.

2.2 Threat Model

We continue with an introduction of the threat model underlying CoRReCt. Looking ahead, we propose a formal model capturing this informal threat model in Section 2.4.

CoRReCt considers the execution of software on commodity hardware, say on the Intel x86-64 architecture in the presence of an adversary. In line with the standard adversarial model in cryptography, CoRReCt considers intelligent adversaries. In particular, the only restriction we impose on adversaries is that their computational power is polynomially bounded. We distinguish between the system (*i.e.* the hardware including its firmware), and the application (*i.e.*

software deployed by the user, including the operating system, system libraries, . . .). CoRReCt can also be applied to executions where this distinction is made on a different abstraction level ("System" up to OS, "Application" all userspace libraries; "System" up to Java Virtual machine, "Application" all Java Code), but for the rest of this work we will focus on the x86-64 machine as dividing line.

Guarantees of an Honest Execution. First, we consider the expected security guarantees of software execution on a single machine in the setting outlined above.

Even if software and hardware are uncorrupted, the execution of software is subject to non-determinism at various levels, for example due to external input and output, network messages, jitter or scheduling decisions of the operation system. In contrast to randomness, which is usually assumed to be of high entropy, no guarantees with respect to the behavior or distribution of non-determinism is made. Consequently, even "unexpected" non-deterministic behavior should not negatively affect the execution. (Of course, in the absence of a formal specification, none of this is well-defined.) The fact that software and hardware are assumed to be uncorrupted does not rule out that they contain bugs that lead to undesirable behavior from the perspective of the user, for example incorrect computation results. Also, the software and hardware may be subject to vulnerabilities that can be exploited by adversaries, resulting in corrupted software or hardware that is discussed below.

Guarantees of an Execution on a Single Corrupted System. Clearly, if hardware or software are corrupted, little security guarantees remain. If only the software is corrupted, for example due to the exploitation of vulnerabilities, the adversary may learn all secrets and perform arbitrary changes to the execution. Still, hardware-based security may remain, for example the privacy of keys stored in a trusted platform module (TPM). Also, if hardware and firmware are uncorrupted, the system may be returned to an honest state by reinstalling the software.

Conversely, if the hardware is corrupted, the situation is even worse, because corrupted hardware may deviate arbitrarily from the software execution, and security supposedly provided by the hardware are lost as well. In particular, corrupted hardware could leak stored cryptographic keys or unfaithfully execute hardware-based cryptographic protocols, for example for remote attestation. Also, recovering from corrupted hardware may be much harder. Even if only the firmware is corrupted, physical access with special tools may be needed.

Adversarial Model of CoRReCt. In contrast to the previous two paragraphs, which considered executions on a single machine only, CoRReCt considers a distributed execution on machines P_1 and P_2 whose results are compared by an input-output interface IO. In more detail, we assume that the machines P_1 and P_2 are commodity hardware running commodity software. We do not assume that there exists a formal specification of correctness, let alone a proof of correctness or security for the software or for P_1 and P_2. Also, we again assume that P_1 and

P_2 are connected to the Internet (via IO). For the sake of simplicity, we assume that all external entities are adversarially controlled.

In this setting, CoRReCt aims to protect an *honest* user against the consequences of corrupted hardware. Other goals such as protecting against the consequences of corrupted *software* are out of scope as discussed below.

Adversaries may corrupt P_1 or P_2 in a number of ways. For example, hardware or firmware of P_1 or P_2 may already be corrupted when deployed by the user, for example due to a supply chain attack. Corruption may also occur after P_1 and P_2 are deployed, for example because an adversary was able to remotely exploit a vulnerability or get physical access to a machine. CoRReCt aims to protect against the consequences of corrupted hard- and firmware as long as only one of P_1 and P_2 is corrupted and IO is honest. While there CoRReCt cannot *prevent* corruption, we believe that this corruption model is justified by the fact that a) the honest user is able to prevent physically tampering of the hardware once deployed, b) by (anonymously) sourcing commodity hardware from different sources, the risk of supply chain attacks can be reduced and, finally, c) due to the use of heterogenous systems, the risk that *both* manufacturers produce corrupted systems is reduced.[3]

The focus of CoRReCt on hard- and firmware corruptions is due to the fact that they may be very hard to prevent, detect or fix. This is because hardware and firmware often are closed-source, making it impossible to judge whether it is uncorrupted or not. Even reliably determining whether a certain firmware is used is often very hard in practice. Thus, it is desirable to at least mitigate their negative consequences as much as possible.

CoRReCt cannot and does not want to protect against the consequences of corruption of the software. Indeed, if the software is corrupted, few if any security guarantees may be left. However, we believe that the problem is more easily solved. To this end, tools like memory-safe programming languages or even formal verification are available. However, if hard- or firmware are corrupted, even software that is formally verified for correctness and the absence of vulnerabilities may not provide any benefits, because the hardware can deviate from the software's execution in an arbitrary way.

Adversarial Influence in CoRReCt. In a setting where the assumptions about the adversarial model hold, the adversary's influence is limited as follows:

– It can arbitrarily interact with the software, possibly exploiting vulnerabilities.
– It can arbitrarily control the hardware of the corrupted system.
– It can arbitrarily control the timing of the execution if either machine is corrupted.
– It can force the stop of the execution either directly on the corrupted system or by presenting an output of the corrupted system that does not match the output of the honest system. In this case, IO will abort.

[3] While not formally considered, CoRReCt may even offer security guarantees if both P_1 and P_2 are corrupted, but do not cooperate. This may, for example, be the case if the corruptions originated from state-level attackers of hostile states.

– Finally, it can arbitrarily control the non-determinism exposed to the software if the *recording* machine P_1 is corrupted.

While the first four points are clear and natural, the last point merits further discussion.

Impact of Non-Determinism. To illustrate the (perhaps surprising) consequences of (adversarially-controlled) non-determinism in the execution, consider the following example. Let p be a program that does the following: Repeatedly sample a bit b using sources of non-determinism. If $b = 0$, read k random bits, for example through a source of randomness provided by the operating system, but do not generate output. If $b = 1$, also read k random bits and output them. This program illustrates the influence of non-determinism, *e.g.* on a system with multiple processes and a preemptive scheduler between them where one process is the application we are interested in (when $b = 1$), and the other is some irrelevant processes, requiring randomness as well (when $b = 0$).

Suppose that the adversary controls the recording machine of CoRReCt. In this case, it is aware of the system's randomness and may fully choose the bit b each time. Let us assume the adversary to choose $b = P(r)$ as a general predicate of the read bits $r \in \{0,1\}^k$. Let $p := \Pr[P(r) = 1]$, for $r \in \{0,1\}^k$ chosen uniformly at random. In this case the expected number of non-deterministic choices before an output is made is: $p \cdot 1 + p(1-p) \cdot 2 + p(1-p)^2 \cdot 3 + \ldots = \frac{1}{p}$.

The value of the chosen r is uniformly distributed among all $n := p \cdot 2^k$ possible values with $P(r) = 1$. Thus, the entropy for an adversary knowing the sampling procedure is $\sum_{i=0}^{n} -\frac{1}{n} \log(\frac{1}{n}) = \log(n) = k + \log p$ and an attacker spending additional time t can reduce the entropy in the randomness by $\log(t)$.

This relation illustrates how important it is to obtain randomness from a shared generator in large enough blocks k, as this sets the entropy baseline that the attackers runtime is weighted against. Also, it illustrates the perhaps surprising influence of adversarially-chosen non-determinism of an otherwise honest execution. An obvious consequence of such a setting is that programs should not assume that the non-determinism is "benign", but should be able to deal with the worst case.

For an application that requires randomness to obtain *e.g.* a cryptographic key inside CoRReCt, it is vital, that it gets its output from the system PRNG in *one* block instead of several smaller blocks. Tinfoil Chat, one of the applications we will present later in Section 3.2, obtains its randomness in one block[4].

2.3 Corruptions and Non-Determinism

Consistent with the problem we intend to address, namely the lack of trust in today's systems, and in line with our threat model (Section 2.2), we consider different levels of adversarial influence on the execution of μ: Even if all entities

[4] `https://github.com/maqp/tfc/blob/07a819b3a8e1ce98bfc292b0ee8a76cb713e9645/src/common/crypto.py#L709`

are honest, we consider non-deterministic aspects of the execution of μ to be under adversarial control. To this end, the adversary is allowed, for every machine μ' that is part of the execution, to specify a special (efficient) Turing machine μ'_N. μ'_N has access to all state of μ' and may, depending on this state, determine non-deterministic parts of the execution. Intuitively, this captures the freedom a machine implementation has when choosing which non-deterministic option to take. This is consistent with our expectation that "useful" programs need to be able to deal with arbitrary non-determinism that may influence their execution. For a program not affected by non-determinism, this trivially guarantees a correct execution (if the machine executing the program is honest). Still, a program might often employ actions that are non-deterministic. Examples include user input or output or network communication. Then, the program must be able to deal with the resulting non-determinism. In particular, this mechanism allows to capture leakage of secrets through non-deterministic aspects of an execution if μ (*i.e.* the program) and μ_N are designed appropriately (see Appendix C).

If P_1 is corrupted, we assume that the adversary learns the inputs of P_1, the code of μ as well as the random tape of μ. Also, in case of corruption, P_1 is fully controlled by the adversary and may deviate from its specified behavior arbitrarily. Additionally, the adversary may *adaptively* and arbitrarily determine non-deterministic parts of the execution. In contrast to the influence on non-corrupted machines, an adversary can make a corrupted machine deviate arbitrarily from the machine specification and choose its output.

If P_1 is corrupted but P_2 is honest, we can prove that the only possibility for the adversary to influence the execution of μ (except for aborts) is to *adaptively* determine the non-deterministic parts of its execution: If a corrupted P_1 deviates from its protocol in a way that influences the (presumptive) outputs of μ, then the honest party P_2 will produce a different output that is detected by IO, leading to a halt of the execution.

Conversely, if P_2 is corrupted but P_1 is honest, the guarantees are very similar to the previous case, with the exception that the non-deterministic parts of the execution are governed by μ_N of P_1 and not adaptively by the adversary.

We assume that IO is honest at all times. This is motivated by the fact that the functionality of IO is very simple and may be implemented in a trusted and verifiable way, for example as a fixed-function circuit.

2.4 Model

In order to formally capture the security provided by our approach, a model is needed. To this end, we use a simplified variant of the Universal Composability (UC) framework [20].

The basic machine model of UC is a so-called *interactive Turing machine*. We retain this machine model, but augment it to model non-determinism of program execution encountered in the real world. To this end, we introduce an additional *non-determinism tape* that can be accessed read-only. We call the resulting machine model *interactive Turing machines with non-deterministic events*, formally defined as follows:

Definition 1 (Interactive Turing Machine with Non-Deterministic Events). *An* interactive Turing machine with non-deterministic events *is defined like an interactive Turing machine [20] with an additional read-only* non-determinism tape. *This tape is initiated with uniform randomness.*

We stress that, as discussed above, the non-determinism tape does *not* model non-deterministic Turing machines. In particular, the complexity class we consider remains $\mathcal{BPP}$. The tape's purpose is to provide an explicit mechanism for possibly adversarially controlled non-determinism.

For a (real) program under analysis, *all* non-determinism that is to be considered during the execution must be modeled through behavior depending on the content of the non-determinism tape.

We define the *non-determinism trace* of an activation of a Turing machine as the sequence of symbols read from the non-determinism tape.

Definition 2 (Non-Determinism Trace). *Let c_1 be the configuration of an (interactive) probabilistic Turing machine with a non-determinism tape. Upon activation on some input that ends in configuration c_2, the* non-determinism *trace t is the sequence of symbols read from the non-determinism tape.*

We consider the following entities: An interactive distinguisher $\mathcal{Z}$, called *environment*, that interacts with an adversary $\mathcal{A}$ as well a protocol π comprised of "main parties" P_1, P_2 and IO. Looking ahead, the task of $\mathcal{Z}$ will be to distinguish between an execution of a protocol π_1 and an adversary $\mathcal{A}$ and a protocol π_2 and an adversary $\mathcal{S}$ called the *simulator*.

Informally, the simulator's task is to simulate the execution of π_1 and $\mathcal{A}$ for $\mathcal{Z}$, even if π_2 is actually executed. Usually, π_2 will be a protocol that is secure by definition. If the indistinguishability of these two executions can be shown, then all properties guaranteed by π_2 also hold in an execution of π_1.

2.5 Execution Experiment

We now define the execution experiment. This execution captures the setting described in Section 2.1 and is closely modeled after the UC execution[5].

Definition 3 (Execution Experiment). *Let $\mathcal{Z}$ and $\mathcal{A}$ be polynomial-time interactive Turing machines. Let P_1, P_2 and IO be polynomial-time interactive Turing machines with non-deterministic events. Let π be a protocol with main parties P_1, P_2 and IO.*

We consider the following execution with machines $\mathcal{Z}$, $\mathcal{A}$ and a protocol π.

1. *On input $(1^\kappa, z)$ for the execution, a machine called* environment *is invoked with the code of $\mathcal{Z}$ and input $(1^\kappa, z)$.*

[5] The changes to incorporate non-determinism would require to re-prove all properties of UC security. As we do not consider composable security, we consider this to be out of scope for this contribution. Instead, we craft a custom model based on UC without establishing possible properties.

2. *On its first activation, the environment invokes the adversary, which is given the code of $\mathcal{A}$ and an input chosen by the environment.*
3. *On its first activation, the adversary may corrupt an arbitrary subset of parties in the set $\{P_1, P_2\}$. Corrupted parties are under full adversarial control and may deviate from the protocol arbitrarily. Let $\mathcal{C}$ denote the set of corrupted parties. For each honest party P in the set $\{P_1, P_2, IO\} \setminus \mathcal{C}$, the adversary may specify the description of a probabilistic polynomial-time Turing machine μ_P that adaptively provides the values on the non-determinism tape of P relative to the code and contents of all other tapes of P. Then, it activates the environment again.*
4. *The environment may provide input to IO, whose code is specified in π and receive output from IO.*
5. *IO, P_1 and P_2 act as specified in the protocol π, which may entail the creation of further* sub-parties *that behave as specified in the protocol.*
6. *The environment and the adversary may communicate with each other at any point. Moreover, the adversary may participate in the execution on behalf of corrupted parties.*
7. *The environment eventually outputs a bit b and halts.*
8. *Unless noted otherwise, all communication between honest entities is ideally secure and immediate,* i.e. *oblivious for the adversary.*

2.6 Ideal Functionality

With the execution experiment at hand, we define an ideal functionality $\mathcal{F}$ that captures the security guarantees provided by our approach. By definition, $\mathcal{F}$ is incorruptible and parameterized with a program μ in the form of the code of a probabilistic polynomial-time Turing machine. Also, $\mathcal{F}$ interacts with parties IO, P_1 and P_2 as well as an adversary.

Initially, $\mathcal{F}$ samples a new random tape for μ. Then, μ is executed inside $\mathcal{F}$ based on inputs provided by IO to $\mathcal{F}$. Outputs of μ are done through IO. Depending on which parties are honest or corrupted, the adversary is able to influence the execution or learn information in several ways:

– If all entities are honest, the adversary may provide a machine μ_N that provides the content of the non-determinism tape for μ, depending on μ's current state. However, the adversary neither learns inputs or outputs, nor the randomness of μ through $\mathcal{F}$.
– If only either P_1 or P_2 are corrupted, the adversary learns the inputs and randomness of μ. If P_1 is corrupted, the adversary may *adaptively* choose the contents of μ's non-determinism tape.
– If both P_1 and P_2 are corrupted, there are essentially no guarantees.

We assume that IO is always honest.

Definition 4 (Ideal Functionality $\mathcal{F}$ for CoRReCt). *Parameterized by a TM μ with non-determinism tape with alphabet Γ, parties P_1, P_2, IO and a security parameter κ.*
Initially, set state $= \bot$, initialize an empty list $\mathcal{L}$ and sample a fresh random

tape for μ.

Corruption and Non-Determinism.

- *If P_1 and P_2 are honest, let the adversary provide the description of a probabilistic polynomial-time Turing machine μ_N. Whenever μ reads from its non-determinism tape, execute μ_N on the current configuration of μ, resulting in output $\gamma \in \Gamma$. Report γ to μ as the current symbol on its non-determinism tape.*
- *If at least one P_i is corrupted (for $i \in \{1, 2\}$), send the contents of the random tape and the description of μ to the adversary. Also, whenever μ attempts to read a symbol from its non-determinism tape and P_1 is corrupted, send* $(\mathtt{query})$ *to the adversary and receive* $(\mathtt{answer}, \gamma)$ *with $\gamma \in \Gamma$. Provide μ with γ as the symbol read from the non-determinism tape.*
- *If both P_1 and P_2 are corrupted, let the adversary determine all outputs.*
- *On message* $(\mathtt{query} - \mathtt{io})$ *from the adversary:*
  - *If at least one P_i is corrupted, send* $(\mathtt{query} - \mathtt{io}, \mathcal{L})$ *to the adversary.*
  - *Otherwise, send* $(\mathtt{query} - \mathtt{io}, \perp)$ *to the adversary.*

Execution.

- *On input* $(\mathtt{input}, m)$ *for IO, add m to $\mathcal{L}$ and execute μ on input* $(1^\kappa, state, m)$*. Let m' be the output of μ and let $state'$ be the state after the execution. Save $state = state'$, add m' to $\mathcal{L}$ and generate a private*
  - *output* $(\mathtt{output}, m')$ *if P_1 and P_2 are honest resp.*
  - *delayed output* $(\mathtt{output}, m')$ *if P_1 or P_2 is corrupted*

  for IO. Accept further inputs.

As a consequence, the security guarantees provided by $\mathcal{F}$ are only as good as the security guarantees of μ are relative to the considered model of execution. In particular, they depend on the influence non-determinism has on the output(s) of μ. In particular, μ_N can possibly be used to exfiltrate secrets, even if P_1 and P_2 are honest. For a given program to be analyzed, determining the right amount of non-deterministic influence in μ and the behavior of μ_N is up to the analyst.

In the following, we define the protocol $\pi_\mathcal{F}$ that wraps an execution of $\mathcal{F}$, similar to an *ideal protocol* in the UC framework [20]:

Construction 1 (The protocol $\pi_\mathcal{F}$) *Let $\pi_\mathcal{F}$ be the following protocol for the parties P_1, P_2 and IO, parameterized with a Turing machine μ. Let $\mathcal{F}$ be the ideal functionality of Definition 4.*

- *P_1 and P_2: Ignore all inputs and all outputs. In case of corruption, inform $\mathcal{F}$ of the corruption.*
- *IO:*
  - *On input* $(\mathtt{input}, m)$*, send input* $(\mathtt{input}, m)$ *to $\mathcal{F}$.*
  - *On output* $(\mathtt{ouptut}, m')$ *from $\mathcal{F}$, give output* $(\mathtt{output}, m')$*.*

As P_1, P_2 and IO are deterministic by definition in this idealized execution, we do not need to consider their non-determinism tapes.

2.7 Protocol

We now state a protocol that captures the approach presented in Section 2.1 in our model. Informally, π works as follows: On the first input (of IO), P_1 and P_2 call the ideal functionality for coin-tossing $\mathcal{F}_{ct}$ (Definition 5) to obtain a uniformly random string, which is used as the random tape for the execution of the TM μ. P_1 acts as recording machine executing μ and evaluating all non-deterministic events by reading from its non-determinism tape. These non-determinism traces are subsequently provided to P_2 in order to be able to replay the execution. Both P_1 and P_2 provide their outputs to IO. If both outputs match, IO generates an output. Conversely, IO also distributes inputs to P_1 and P_2. For honest IO, P_1 and P_2, we assume that they directly communicate without adversarial influence.

Construction 2 (The CoRReCt Protocol π) *π is parameterized with a probabilistic polynomial-time TM μ. The main parties of π are P_1, P_2 and IO.*

Protocol of IO:

1. *Initiate $o_1, o_2 = \bot$.*
2. *On input $(\texttt{input}, m)$:*
   - *If this is the first input, send $(\texttt{coin}-\texttt{toss})$ to P_1. After having received $(\texttt{coin}-\texttt{toss})$ from P_1, and for every subsequent input, continue as follows:*
   - *Send $(\texttt{input}, m)$ to P_1 and receive $(\texttt{output}, m_1)$ from P_1. Activate P_1 again.*
   - *Send $(\texttt{input}, m)$ to P_2 and receive $(\texttt{output}, m_2)$ from P_2.*
   - *After having received $(\texttt{output}, m_i)$ from both P_1 and P_2: If $o_i = \bot$, set $o_i = m_i$. If $o_1 = o_2 \neq \bot$, output $(\texttt{output}, o_1)$ and set $o_1 = o_2 = \bot$. If $o_1 \neq o_2$ and $o_1 \neq \bot$ and $o_2 \neq \bot$, halt.*

Protocol of P_1:

1. *On message $(\texttt{coin}-\texttt{toss})$ from IO:*
   (a) *Send $(\texttt{coin}-\texttt{toss})$ to $\mathcal{F}_{ct}$.*
   (b) *Receive $(\texttt{coin}-\texttt{toss}, r)$ from $\mathcal{F}_{ct}$. Store r and use r as randomness for the execution of μ.*
2. *On message $(\texttt{input}, m)$ from IO:*
   (a) *Set $t = \varepsilon$.*
   (b) *Continue the execution of μ on input m, recording a non-determinism trace t.*
   (c) *Store the current state of μ and send $(\texttt{output}, m')$ to IO, where m' is the output of μ.*
   (d) *On the next activation, send $(\texttt{trace}, t)$ to P_2.*

Protocol of P_2:

1. *On message $(\texttt{coin}-\texttt{toss})$ from IO:*
   (a) *Send $(\texttt{coin}-\texttt{toss})$ to $\mathcal{F}_{ct}$.*
   (b) *Receive $(\texttt{coin}-\texttt{toss}, r)$ from $\mathcal{F}_{ct}$. Store r and use r as randomness for the execution of μ.*

2. On message $(\texttt{trace}, t)$ *from* P_1*, store* t *and activate* IO*.*
3. On message $(\texttt{input}, m)$ *from* IO*:*
   (a) Continue the execution of μ *on input* m *and trace* t *for the non-determinism tape. If there is no stored trace* t*, halt.*
   (b) Delete t*, store the current state and send* $(\texttt{output}, m')$ *to* IO*, where* m' *is the output of* μ*.*

We now state our main theorem, namely that our protocol π realizes the ideal functionality $\mathcal{F}$ for CoRReCt. Thus, π is "just as secure" as the ideal functionality $\mathcal{F}$, meaning that all security guarantees of $\mathcal{F}$ also hold in an execution with π.

Theorem 1. *For every PPT adversary* $\mathcal{A}$*, there exists a PPT simulator* $\mathcal{S}$ *such that for every PPT environment* $\mathcal{Z}$*, the output of* $\mathcal{Z}$ *in the execution of* π *with* $\mathcal{A}$ *and the execution of* $\pi_{\mathcal{F}}$ *and* $\mathcal{S}$ *are identically distributed.*

For the proof, see Appendix B.

3 Applications

We discuss a series of applications of the CoRReCt concept that vary in their security requirements.

3.1 Secure Multi-Party Computations

In a setting where the strong security guarantees of secure multi-party computations (MPC) are desirable, using CoRReCt to participate in the protocol offers an interesting trade-off: On the one hand, if at least one system performing the party's computation is honest and an output is returned, CoRReCt guarantees that this output is the actual output of the computation. On the other hand, CoRReCt may negatively impact the privacy: In a k-round protocol, $\log_2(k)$ bits can be leaked simply by one of the systems aborting at a point of its choice.

Additionally, a malicious recording system may use timing side-channels to leak information. If the same (malicious) system would have been used to participate in the computation when not using CoRReCt, there would not have been any security guarantees. In particular, it could use different inputs for the computation, present different outputs and leak *all* secrets to the adversary. Thus, under this corruption, the timing side-channel that still exists with CoRReCt may not constitute a loss of security. However, if the recording system were honest, a malicious replaying system could still leak information by aborting, which constitutes a loss in security. In order to protect from timing side-channels of the replaying system, outputs must be time-synchronized: If the recording system wants to output a value x at time t, this output may only be allowed at time $t + T$ if the replaying system also outputs x at time t' in the interval $(t, t + T)$. This reduces the possible influence of the replaying system to aborts at the expense of additional latency governed by the parameter T.

For certain cases, better guarantees may be obtained: If the party has no private input (*e.g.* because it is the receiver of a commitment) and its output is computationally hidden until the last message is received, the leakage can be reduced to a single bit. This even works in case of a corrupted recording system if a timeout for returning the result after receiving the last message is enforced.

3.2 Tinfoil Chat

Tinfoil chat [6] aims to provide a high-security solution for secure communication. To this end, four dedicated systems are used: A *source computer* which receives outgoing messages from the user and is connected to a *data diode* such that it can only send messages, but not receive them. In particular, the source computer is not directly connected to the network. Conversely, a *destination computer* handles incoming messages. It is also attached to a data diode, but in the opposite direction such that it can only receive messages, but not send them. All network communication is handled through a *networked computer*.

The rationale behind this architecture is as follows: If the source computer is initially honest, it cannot be compromised via the network. Thus, plaintext messages are protected from hacks. While the destination computer can receive messages and is thus susceptible to hacks, the data diode prevents exfiltration of keys used for decryption as well as decrypted plaintexts. Note that integrity is not guaranteed in case of corruption. As the networked computer does not handle secrets, it may be corrupted without affecting security.

Taking a closer look, we observe the following: The source computer repeatedly generates cryptographic key material. To this end, it relies on the Linux `getrandom` syscall, which is fed from several sources of non-determinism, which is extensively discussed in the Tinfoil chat code. Thus, Tinfoil chat makes an *implicit* assumption about this non-determinism to be of sufficient quality. If the quality were insufficient, security might be affected. If the source computer were malicious, it could modify the execution to leak secrets, *e.g.* through the selection of randomness of ciphertexts such that the least significant bit of the ciphertext leaks information about key material or plaintexts.

While these problems are out of scope for Tinfoil chat (which explicitly states that it does not protect from "interdiction of hardware"), they are in the scope of CoRReCt. In particular, if CoRReCt is used for the source computer, it can ensure that a) the adversarial influence with respect to the choice of the randomness is limited and b) that malicious hardware or a malicious host operating system could not manipulate the execution. Given that Tinfoil chat also protects (to some extent) from timing side channels through network messages, the level of protection resulting from the combination of both approaches promises to be very strong. We also note that the computations performed by Tinfoil chat are not very computation-heavy. Looking ahead at the measured overhead of our approach, we consider the degradation of performance acceptable, in particular if one deems it necessary to use a system like Tinfoil chat in the first place.

The data diode needs to be extended to also compare the output of the two executions, and to copy the keyboard input to for both source computers. For a concrete implementation a RP2040 microprocessor could be used to compare the contents of two UART streams (which are used by the proposed DIY data diodes), and if equal, forward them to the data diode. Using its USB interface a USB keyboard can be read out and fed to both source systems equally via the USB-UART converters. As input from the keyboard will be be echoed on both source computers, a corrupt keyboard cannot change the input unnoticed. The user would need to keep an eye on both source computers' displays before relying on the messages displayed there. Care needs to be taken that the code running the comparison cannot be overridden *e.g.* from the two source computers.

To test if the performance of CoRReCt is sufficient to run Tinfoil chat, we started a test installation of all Tinfoil chat components within CoRReCt. For inputs into CoRReCt we observed an input delay of approximately 200ms on average, being noticable, but not hindering usage. Sending messages with Tinfoil chat via TOR already incurs some latency so that additional latency introduced by CoRReCt was not noticable.

3.3 Access Control

Evaluating access control policies can be complex and a high value target for an attacker. When the output of this application is deterministic, CoRReCt can be used to ensure the integrity of the resulting access decision. However, a third party may need some way to authenticate the output of this computation. Using a signing key, as in the general server application, requires the elimination of non-determinism both on protocol level (*e.g.* TLS or something simpler) as well as on application level to prevent unintended leakage of cryptographic secrets.

3.4 Software Building

Systems used to compile software are a valuable target for supply-chain attacks. Reproducible builds[6] make software build processes deterministic by removing any influence non-determinism might have on the artifacts, greatly limiting an adversary's influence. When the build process cannot be adjusted, there are scientific approaches to reduce non-determinism to some extent [15], but there are still many special cases that are not handled. In case of proprietary software, the build process can be very complex, as it might involve the usage of closed-source tools, automatic dependency downloads and other software that makes the process non-deterministic without the possibility to troubleshoot difficult. In this case, CoRReCt can help to verify the remaining non-determinism in a build process. What is more, the event stream and build system image can be kept on record to allow to re-verify the built artifacts later for audit purposes.

Table 1 shows a high-level comparison of different approaches to securing software building. SGX and SEV as commercially available enclaves aim to ensure correctness and integrity by relying on a vendor created key. AMD SEV and Intel TDX, as concepts that target VMs, are more general and heavyweight, while SGX focuses on isolating individual applications. CoRReCt, in line with AMD SEV and Intel TDX, allows ensuring the correctness of a whole VM execution and is therefore the most generally applicable.

[6] `https://reproducible-builds.org`

	Repr. Builds	SGX	SEV/TDX	CoRReCt
Performance	✓	✓	✓	TCG overhead single-threaded
Auditability	✓	×	×	✓
Security not based on HW trust	✓	×	×	1-of-2 + simple comparator
Applicable to generic build process	×	?	✓	✓

Table 1. High-level comparison of the advantages and disadvantages of the different approaches for software building.

4 Realization of CoRReCt with QEMU Linux VMs

In this section, we present a realization of our approach, with QEMU as a machine emulator at its core. In more detail, we consider the execution of Linux VMs for the x86-64 architecture with remote network access, motivated by the "Tinfoil Chat" and "software building" use-cases presented in Section 3. To this end, we modified both QEMU as well as the Linux kernel as outlined below.

We rely on QEMU's record-replay for deterministic replay by first recording non-deterministic events during the execution of a virtual machine. These events are written to a file and used as input during replay. Technically, an event consists of an ID to determine what kind of event was recorded, together with associated data. Some examples are network events, clock events and interrupt events. To be able to replay, a recording QEMU VM additionally stores the number of executed instructions in the event log. During replay, this number is used to inject recorded events at the right moment.

By default, record-replay first records an execution and replays it after the recording has finished. For CoRReCt, we modified QEMU to enable the "pseudo-parallel" execution of recording and replaying. To this end, two separate QEMU instances are created at the same time—one for recording and one for replaying. When the recorder starts the execution, it sends all events directly to the replayer. The replayer immediately replays the recording by using the event stream as input. We call this pseudo-parallel because of the delay for transmitting events.

Providing Randomness. To provide randomness that is suitable for cryptographic applications and *not* generated by harvesting non-determinism, it was necessary to implement changes to a) Exchange a seed for initializing a random number generator (RNG), b) pass a stream of random bytes from this RNG to QEMU, and c) modify the Linux kernel's RNG to only use the provided randomness.

Passing randomness to QEMU is possible by specifying a file to read from. This file's contents are forwarded to the VM and presented as a hardware RNG, accessible from userspace through `/dev/hwrng`. Linux' RNG uses this device file by default as source for its entropy pool. On top of that, it uses additional sources of randomness that are not suitable for our purpose because they rely on non-determinism, *e.g.* interrupt timings. As the values of theses sources can be set unilaterally by the recording VM, we modified the Linux kernel to only use `/dev/hwrng` as source of randomness. The generation of a suitable random stream happens outside of QEMU. This is done by exchanging a seed for a RNG between the recorder and replayer based on the coin tossing protocol of Blum [21]. After the exchange is finished, the seed is used to initialize a PRNG.

Handling Network Traffic. To handle network traffic, there exists a third system that is connected to both QEMU instances and is called *input-output interface*. Those connections are established by using QEMU's socket network backend. The interface's first task is to send incoming packets from the network to both virtual machines. Its second task is to receive outgoing packets from both virtual machines, compare them and, if they match, forward them to the network. If outgoing network packets do not match, execution is stopped. To enable the usage of the interface, QEMU was modified such that network packets can be injected and sent during replay. The default settings for replay mode discards outgoing packets and reads incoming packets from the event log file. Enabling transmission of outgoing packets required changes to QEMU's network filter for record-replay such that outgoing packets are not discarded. Packet injection is achieved by storing network packets sent by the interface in a separate list inside of QEMU. This list is then used whenever a packet is read from the event log to replace the network packet from the log with the one injected by the interface. An additional property of this system is that network traffic is inherently ordered because of deterministic replay.

Benchmarks. Tables 2 and 3 show benchmarking results for three different scenarios on two machines with Intel Xeon D-1718T CPUs. First, QEMU is run with KVM, *i.e.* kernel-based hardware-assisted virtualization, to establish a base line. Second, QEMU only uses software virtualization with single-threaded TCG, *i.e.* the QEMU tiny code generator that translates instructions of the guest architecture to instructions of the host architecture, which is a prerequisite for CoRReCt. The third scenario is CoRReCt, TCG run with record-replay and accessed through a comparator system. Tables 2 and 3 show the results of an I/O-heavy benchmark[7] using the Flexible IO Tester resp. the results of a CPU-heavy benchmark[8] that uses FFmpeg. The exact commands and dependencies used to execute these benchmarks in the VM can be found in the linked files.

In both cases, there is a significant performance decrease when disabling KVM, which is expected due to the overhead of TCG. Compared to TCG execution, using CoRReCt has a significant impact on I/O. On an abstract level, this stems from the fact that I/O timing involves a lot of non-deterministic behavior that has to be recorded and replayed. Optimizing parts of QEMU to handle this more efficiently could narrow this performance gap. Computation itself is only very slightly impacted by CoRReCt in comparison to TCG execution as it is mostly deterministic and the main overhead stems from the effort of re-playing events. To

[7] `https://openbenchmarking.org/test/pts/fio` (Random Read/Write, Sync, No Direct, 8MB Block Size)

[8] `https://openbenchmarking.org/test/pts/ffmpeg` (h264, Live)

KVM	508 MB/s ± 1.20 %
TCG	155 MB/s ± 2.48 %
CoRReCt	23.7 MB/s ± 30.10 %

Table 2. Average writing speed across three runs FIO with CoRReCt and compared to pure TCG and KVM as baseline

KVM	6.77 FPS ± 0.41 %
TCG	0.21 FPS ± 0.45 %
CoRReCt	0.20 FPS ± 0.24 %

Table 3. Average achieved frames per second (higher numbers are better) across three runs for encoding video into h264 with ffmpeg. Then Benchmark is run with CoRReCt and compared to pure TCG and KVM as baseline

capture this accurately, the benchmark was extended to wait for a synchronization point after the computation finished, ensuring that the CoRReCt system is completely done processing this benchmark. The conclusion drawn from these results is that the main cost associated with using the CoRReCt system stems from running QEMU in TCG mode, especially for computation-heavy workloads. For I/O-bound tasks, CoRReCt incurs overhead due to synchronization. Reduced performance compared to hardware virtualization is an expected outcome, but necessary if different host architectures are to be used.

5 Conclusion

We presented an approach to improve the security and privacy of software executions on untrusted systems. To this end, we used virtual machines to execute the software, where non-deterministic events are recorded on one and replayed on another VM. Cryptographically-secure randomness is generated through a coin-toss, replacing randomness-harvesting methods that rely on non-determinism and may negatively impact security. Inputs and outputs are handled by a trusted interface and only allowed if they are the same by both VMs.

We provided a formal model for our approach, together with a proof of security in this model. Assuming a trusted input-output interface, our approach leads to an execution with strong security guarantees:

- If at most one system is corrupted, the correct inputs are used and either a correct result or no result at all is returned.
- If at most one system is corrupted and timing side-channels are not considered, then the additional leakage introduced by our approach is bounded by $\log_2(n)$ bits, where n is the number of messages sent. If the recording system is honest and timing side-channels are considered, the same bound can be established.

Given that the VMs can be run on different host platforms, we believe that this assumption is very plausible. Special care was taken to model an adversary's influence on an otherwise honest execution through its choice of non-determinism.

We discussed several practically relevant applications of our approach and presented our implementation based on QEMU Linux VMs, demonstrating the practicality of CoRReCt.

References

1. Goldreich, O., Micali, S., and Wigderson, A.: How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In: Aho, A.V. (ed.) Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA, pp. 218–229. ACM (1987). DOI: 10.1145/28395.28420
2. Yao, A.C.: How to Generate and Exchange Secrets (Extended Abstract). In: 27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986, pp. 162–167. IEEE Computer Society (1986). DOI: 10.1109/SFCS.1986.25
3. Challener, D., Yoder, K., Catherman, R., Safford, D., and Van Doorn, L.: A practical guide to trusted computing. Pearson Education (2007)
4. Costan, V., and Devadas, S.: Intel SGX explained. Cryptology ePrint Archive (2016)
5. Scales, D.J., Nelson, M., and Venkitachalam, G.: The design of a practical system for fault-tolerant virtual machines. ACM SIGOPS Operating Systems Review 44(4), 30–39 (2010)
6. MISC
7. Iturbe, X., Venu, B., Ozer, E., and Das, S.: A triple core lock-step (TCLS) ARM® Cortex®-R5 processor for safety-critical and ultra-reliable applications. In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W), pp. 246–249 (2016)
8. Lamport, L., Shostak, R.E., and Pease, M.C.: The Byzantine Generals Problem. ACM Trans. Program. Lang. Syst. 4(3), 382–401 (1982). DOI: 10.1145/357172.357176
9. Castro, M., and Liskov, B.: Practical Byzantine Fault Tolerance. In: Seltzer, M.I., and Leach, P.J. (eds.) Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999, pp. 173–186. USENIX Association (1999)
10. Ronsse, M., and De Bosschere, K.: RecPlay: A fully integrated practical record/replay system. ACM Transactions on Computer Systems (TOCS) 17(2), 133–152 (1999)
11. Sen, K., Kalasapur, S., Brutch, T., and Gibbs, S.: Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 488–498 (2013)
12. Bressoud, T.C., and Schneider, F.B.: Hypervisor-based fault tolerance. ACM Transactions on Computer Systems (TOCS) 14(1), 80–107 (1996)
13. Zhu, J., Dong, W., Jiang, Z., Shi, X., Xiao, Z., and Li, X.: Improving the performance of hypervisor-based fault tolerance. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–10 (2010)
14. Liu, H., Jin, H., Liao, X., and Pan, Z.: XenLR: Xen-based Logging for Deterministic Replay. In: 2008 Japan-China Joint Workshop on Frontier of Computer Science and Technology, pp. 149–154 (2008). DOI: 10.1109/FCST.2008.31
15. Navarro Leija, O.S., Shiptoski, K., Scott, R.G., Wang, B., Renner, N., Newton, R.R., and Devietti, J.: Reproducible Containers. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '20, 167–182. Association for Computing Machinery, Lausanne, Switzerland (2020). DOI: 10.1145/3373376.3378519
16. Herzberg, A.: Folklore, practice and theory of robust combiners. Journal of Computer Security 17(2), 159–189 (2009)
17. Harnik, D., Kilian, J., Naor, M., Reingold, O., and Rosen, A.: On robust combiners for oblivious transfer and other primitives. In: Advances in Cryptology–

EUROCRYPT 2005: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005. Proceedings 24, pp. 96–113 (2005)

18. Pietrzak, K.: Non-trivial black-box combiners for collision-resistant hash-functions don't exist. In: Advances in Cryptology-EUROCRYPT 2007: 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007. Proceedings 26, pp. 23–33 (2007)
19. Achenbach, D., Müller-Quade, J., and Rill, J.: Universally composable firewall architectures using trusted hardware. In: Cryptography and Information Security in the Balkans: First International Conference, BalkanCryptSec 2014, Istanbul, Turkey, October 16-17, 2014, Revised Selected Papers 1, pp. 57–74 (2015)
20. Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In: 42nd Annual Symposium on Foundations of Computer Science, pp. 136–145. IEEE Computer Society Press, Las Vegas, NV, USA (2001). DOI: 10.1109/SFCS.2001.959888
21. Blum, M.: Coin flipping by telephone a protocol for solving impossible problems. ACM SIGACT News 15(1), 23–27 (1983)

Supplementary Material

A Ideal Functionality $\mathcal{F}_{ct}$

The ideal functionality for coin-toss $\mathcal{F}_{ct}$ is defined as follows:

Definition 5 (Ideal Functionality $\mathcal{F}_{ct}$). *Parameterized with parties P_1, P_2, a length parameter l and an adversary.*

1. *On first input* $(\texttt{coin} - \texttt{toss})$ *from a party $P \in \{P_1, P_2\}$, sample $r \leftarrow \{0,1\}^l$ uniformly at random. Output* $\texttt{coin} - \texttt{toss}, r)$ *to P.*
2. *On subsequent input* $(\texttt{coin} - \texttt{toss})$ *from the other party $P' \in \{P_1, P_2\}$, output* $(\texttt{coin} - \texttt{toss}, r)$ *to P'.*
3. *Ignore all subsequent inputs.*

B Proof

We now prove Theorem 1.

Proof. To this end, we state a simulator. To make the presentation easier, we consider a simulator for the so-called *dummy adversary* [20], which is the adversary that reports all messages it sees to the environment and delivers all messages it receives from the environment. Intuitively, this adversary is the hardest to simulate. For UC security, one can show that the dummy adversary is complete, *i.e.* if there exists a simulator for the dummy adversary, then there also exists a simulator for every other PPT adversary [20]. It is easy to see that this also holds for our model.

Informally, the task of the simulator is to simulate an execution of π and $\mathcal{A}$ for the environment, while actually $\pi_{\mathcal{F}}$ is executed. If the simulator succeeds with this task, then all properties of $\pi_{\mathcal{F}}$ (and $\mathcal{F}$) also hold in π.

Definition 6 (Simulator). *We distinguish between the following cases:*

All Parties Honest Let μ_N be the machine the environment provides to fill the non-determinism tape of P_1. Send the description of μ_N to $\mathcal{F}$.

P_1 Corrupted but P_2 Honest:

1. *Obtain r from $\mathcal{F}$, report* $(\texttt{coin} - \texttt{toss}, r)$ *as output of $\mathcal{F}_{ct}$ for P_1.*
2. *Internally, emulate an instance I_2 of μ, using random tape r.*
3. *On each activation of P_1, send* $(\texttt{leak} - \texttt{io})$ *to $\mathcal{F}$ and get the current input m.*
4. *When necessary, query non-deterministic events from the environment and provide them to $\mathcal{F}$.*
5. *If the environment instructs the adversary to send a different* $(\texttt{trace}, t')$ *message to P_2 (relative to the trace provided to $\mathcal{F}$), execute I_2 on its current state, input m and the trace t'. If the environment does not instruct the adversary to send a different trace, execute the instance I_2 using the non-deterministic events provided to $\mathcal{F}$.*
6. *If the environment instructs the adversary to send a different* `output` *message from P_1 to IO (relative to the output of I_2), do not allow the corresponding output of $\mathcal{F}$.*

P_2 Corrupted but P_1 Honest:

1. *Let μ_N be the machine the environment provides to fill the non-determinism tape of P_1. Send the description of μ_N to $\mathcal{F}$.*
2. *Obtain r from $\mathcal{F}$, report $(\texttt{coin}-\texttt{toss}, r)$ as output of $\mathcal{F}_{ct}$ for P_2.*
3. *Internally, emulate an instance I_1 of μ, using random tape r and μ_N for the non-determinism tape. Send the traces of the execution to P_2 on behalf of P_1 when P_1 would do so in the protocol.*
4. *On each activation of P_2, send $(\texttt{leak}-\texttt{io})$ to $\mathcal{F}$ and get the current input m.*
5. *Execute instance I_1 of μ on its current state with input m, using the non-deterministic events provided by μ_N and randomness r.*
6. *If the environment instructs the adversary to send a different* `output` *message of P_2 to IO (relative to the output of I_1), do not allow the corresponding output of $\mathcal{F}$.*

Both P_1 and P_2 Corrupted:

1. *Whenever IO would make an output in π, make the corresponding output through $\mathcal{F}$.*

By the definition of $\mathcal{S}$, it follows that the two executions are identically distributed and the claim follows.

C Information Leakage Example

Consider a program that evaluates a (deterministic) and efficient function $F(x)$ and in addition samples non-determinism d and outputs $(F(x), d)$. The adversary might now specify μ_N (see Section 2.1) to compute an approximation for the number of steps of $F(x)$ during execution, such that this information is embedded in d. The result now consists of $F(x)$ and d, effectively allowing to capturing many possible side-channel attack scenarios.