HOP: Hardware makes Obfuscation Practical[*]

Kartik Nayak[1], Christopher W. Fletcher[2], Ling Ren[3], Nishanth Chandran[4], Satya Lokam[4], Elaine Shi[5], and Vipul Goyal[4]

[1]UMD – kartik@cs.umd.edu
[2]UIUC – cwfletch@illinois.edu
[3]MIT – renling@mit.edu
[4]Microsoft Research, India – {nichandr, satya, vipul}@microsoft.com
[5]Cornell University – rs2359@cornell.edu

Abstract

Program obfuscation is a central primitive in cryptography, and has important real-world applications in protecting software from IP theft. However, well known results from the cryptographic literature have shown that software only virtual black box (VBB) obfuscation of general programs is impossible. In this paper we propose HOP, a system (with matching theoretic analysis) that achieves simulation-secure obfuscation for RAM programs, using secure hardware to circumvent previous impossibility results. To the best of our knowledge, HOP is the *first* implementation of a provably secure VBB obfuscation scheme in any model under any assumptions.

HOP trusts only a hardware single-chip processor. We present a theoretical model for our complete hardware design and prove its security in the UC framework. Our goal is both provable security and practicality. To this end, our theoretic analysis accounts for all optimizations used in our practical design, including the use of a hardware Oblivious RAM (ORAM), hardware scratchpad memories, instruction scheduling techniques and context switching. We then detail a prototype hardware implementation of HOP. The complete design requires 72% of the area of a V7485t Field Programmable Gate Array (FPGA) chip. Evaluated on a variety of benchmarks, HOP achieves an overhead of $8\times \sim 76\times$ relative to an insecure system. Compared to all prior (not implemented) work that strives to achieve obfuscation, HOP improves performance by more than three orders of magnitude. We view this as an important step towards deploying obfuscation technology in practice.

1 Introduction

Program obfuscation [29, 4] is a powerful cryptographic primitive, enabling numerous applications that rely on intellectually-protected programs and the safe distribution of such programs. For example, program obfuscation enables a software company to release software patches without disclosing the vulnerability to an attacker. It could also enable a pharmaceutical company to outsource its proprietary genomic testing algorithms, to an untrusted cloud provider, without compromising its intellectual properties. Here, the pharmaceutical company is referred to as the "sender" whereas the cloud provider is referred to as the "receiver" of the program.

Recently, the cryptography community has had new breakthrough results in understanding and constructing program obfuscation [21]. However, cryptographic approaches towards program obfuscation have limitations. First, it is well-understood that strong (simulation secure) notions of program obfuscation cannot be realized in general [4] — although they are desired or necessary in many applications such as the

[*]This is a full version of the paper presented at the Network and Distributed System Security Symposium (NDSS), 2017

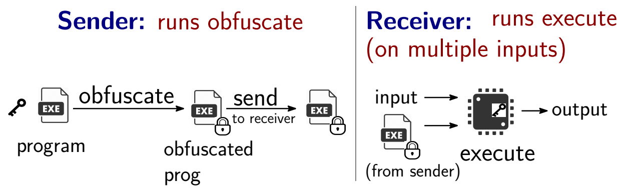

Figure 1: **Obfuscation Scenario.** The sender obfuscates programs using the obfuscate procedure. It sends (possibly multiple) obfuscated program(s) to the receiver. The receiver can execute any obfuscated program with any input of its choice.

aforementioned ones. Second, existing cryptographic constructions of obfuscation (that achieve weaker notions of security, such as indistinguishability obfuscation [22]) incur prohibitive practical overheads, and are infeasible for most interesting application scenarios. For example, it takes ~ 3.3 hours to obfuscate even a very simple program such as an 80-bit point function (a function that is 0 everywhere except at one point) and ~ 3 minutes to evaluate it [37]. Moreover, these cryptographic constructions of program obfuscation rely on new cryptographic assumptions whose security is still being investigated by the community through a build-and-break iterative cycle [14]. Thus, to realize a practical scheme capable of running general programs, it seems necessary to introduce additional assumptions.

In this direction, there has been work by both the cryptography and architecture communities in assuming trusted hardware storing a secret key. However, proposals from the cryptography community to realize obfuscation (and a closely related primitive called functional encryption) have been largely theoretical, focusing on what minimal trusted hardware allows one to circumvent theoretical impossibility and realize simulation-secure obfuscation [27, 15, 17]. Consequently these works have not focused on practical efficiency, and they often require running the program as circuits (instead of as RAM programs) and also utilize expensive cryptographic primitives such as fully homomorphic encryption (FHE) and non-interactive zero knowledge proofs (NIZKs). On the other hand, proposals from the architecture community such as Intel SGX [42], AEGIS [53], XOM [38], Bastion [13], Ascend [18] and GhostRider [40] are more practical, but their designs do not achieve cryptographic definition of obfuscation. In this paper, we close this gap by designing and implementing a practical construction of program obfuscation for RAM programs using trusted hardware.

Problem statement. The problem of obfuscation can be described as follows. A sender, who owns a program, uses an obfuscate procedure to create an obfuscated program. It then sends this obfuscated program to a receiver who can execute the program on inputs of her choice. The obfuscated program should be functionally identical to the original program. For any given input, the obfuscated program runs for time T (fixed for the program) and returns an output.[1] The receiver only has a black box-like access to the program, i.e., it learns only the program's input/output behavior and the bound on the runtime T. In obfuscation, the inputs/outputs are public (not encrypted).

To make use of a trusted secure processor (which we call a *HOP processor*), our obfuscation model is modified as follows (cf. Figure 1). HOP processors are manufactured with a hardwired secret key. The HOP processor (which is trusted) is given to the receiver, and the secret key is given to the sender. Using the secret key, the sender can create multiple obfuscated programs using the obfuscate procedure and send them to the receiver. The receiver then runs the execute procedure (possibly multiple times) to execute the program with (cleartext) inputs of her choice. As mentioned, the receiver (adversary) learns only the final outputs and nothing else. In other words, we offer virtual blackbox simulation security, where the receiver learns only as much information as if she were interacting with an oracle that computes the obfuscated program. In particular, the receiver should not learn anything from the HOP processor's intermediate behavior such as timing or memory access patterns, or the program's total runtime (since each program always runs for a fixed amount of time set by the sender).

Key distribution with public/private keys. We assume symmetric keys for simplicity. HOP may also use a private/public key distribution scheme common in today's trusted execution technology. The obfuscate

[1] T is analogous to a bound on circuit size in the cryptographic literature.

and execute operations can be de-coupled from the exact setup and key distribution system used to get public/private keys into the HOP processor. A standard setup for key distribution [28, 42] is as follows: First, a trusted manufacturer (e.g., Intel) creates a HOP processor with a unique secret key. Its public key is endorsed/signed by the manufacturer. Second, the HOP processors are distributed to receivers and the certified public keys are distributed to senders (software developers). The modification to our scheme in the public key setting is described in Appendix A Note that the key goal of obfuscation is to secure the sender's program and this relies on the secrecy of the private key stored in the processor. Thus, it is imperative that the sender and the manufacturer are either the same entity or the sender trusts the manufacturer to not reveal the secret key to another party.

Non-goals. We do not defend against analog side channels such as measuring power analysis or heat dissipation, we also do not defend against hardware fault injection [8, 3, 34]. We assume that the program to be obfuscated is trustworthy and will not leak sensitive information on its own, including through possible software vulnerabilities such as buffer overflows [7]. There exist techniques to mitigate these attacks, and we consider them to be complementary to our work.

Challenges. It may seem that relying on secure hardware as described above easily 'solves' the program obfuscation problem. This is not the case: even with secure hardware, it is still not easy to develop a secure *and* practical obfuscation scheme. The crux of the problem is that many performance optimizations in real systems (and related work in secure processors [18, 40, 45]) hinge on exploiting *program-dependent behavior*. Yet, obfuscation calls for completely hiding all program-dependent behavior. Indeed, we started this project with a strawman processor that gives off the impression of executing any (or every) instruction during each time step – so as to hide the actual instructions being executed. Not surprisingly, this incurs huge ($\sim 10,000\times$; c.f. Section 3.2) overheads over an insecure scheme, even after employing a state-of-the-art Oblivious RAM [26, 19] to improve the efficiency of accessing main memory. Moreover, in an obfuscation setting, the receiver can run the same program multiple times for different inputs and outputs. Introducing practical features such as context switching — where the receiver can obtain intermediate program state — enables this level of flexibility but also enables new attacks such as *rewinding* and *mix-and-match* execution. Oblivious RAMs, in particular, are not secure against rewinding and mix-and-match attacks and an important challenge in this work is to protect them against said attacks in the context of the HOP system.

1.1 Our Contributions

Given the above challenges, a primary goal of this paper is to develop and implement an optimized architecture that is still provably secure by the VBB obfuscation definition. We stress that *all* the performance optimizations made in the paper are included and proven secure in our theoretic analysis: we want our practical design to match the theory to the extent possible. We view this as an important step towards deploying obfuscation technology in practice.

In more detail, we make the following contributions:

1. Theoretical contributions: We provide the first theoretic framework to *efficiently* obfuscate RAM programs directly on secure hardware. One goal here is to avoid implicitly transforming the obfuscated program to its circuit representation (e.g., [17]), as the RAM to circuit transformation can incur a polynomial blowup in runtime [23]. We also wish for our analysis to capture important performance optimizations that matter in an implementation; such as the use of a cryptographic primitive called Oblivious RAM [25, 26], on-chip memory, instruction scheduling, and context switching. As a byproduct, part of our analysis achieves a new theoretical result (extending [27]): namely, how to provide program obfuscation for RAM programs *directly* assuming only 'stateless' secure hardware.[2] We also show interesting technical subtleties that arise in constructing efficient RAM-model program obfuscation from stateless hardware. In particular, we highlight the different techniques used to overcome all possible forms of *rewinding* and *mix-and-match* attacks (which

[2]Roughly speaking, a HOP processor which allows the host to arbitrary context switch programs on/off the hardware is equivalent to 'stateless' hardware in the language of prior work [27, 15]. This is explained further in Section 3.

may be of independent interest). Putting it all together, we provide a formal proof of security for the entire system under the universally composable (UC) simulation framework [10].

2. Implementation with trusted hardware: We design and implement a hardware prototype system (called HOP) that attains the definition of program obfuscation and corresponds to our theoretic analysis. To the best of our knowledge, this effort represents the *first* implementation of a provably secure VBB obfuscation scheme in *any* model under *any* assumptions. For performance, our HOP prototype uses a hardware-optimized Oblivious RAM, on-chip memory and instruction scheduling (our current implementation does not yet support context switching). As mentioned earlier, our key differentiator from prior secure processor work is that our performance optimizations maintain *program privacy* and exhibit no program-dependent behavior. With these optimizations, HOP performs $5\times \sim 238\times$ better than the baseline HOP design across simple to sophisticated programs while the overhead over an insecure system is $8\times \sim 76\times$. The program code size overhead for HOP is only an *additive* constant. Our final design requires 72% area when synthesized on a commodity FPGA device. Of independent interest, we prove that our optimized scheme always achieves to within $2\times$ the performance of a scheme that does not protect the main memory timing channel (Section 3.3).

2 Related Work

Obfuscation. The formal study of virtual black-box (VBB) obfuscation was initiated by Hada [29] and Barak *et al.* [4]. Unfortunately, Barak *et al.* showed that it is impossible to achieve program obfuscation for general programs. Barak *et al.* also defined a weaker notion of indistinguishability obfuscation ($i\mathcal{O}$), which avoids their impossibility results. Garg *et al.* [22] proposed a construction of $i\mathcal{O}$ for all circuits based on assumptions related to multilinear maps. However, these constructions are not efficient from a practical standpoint. There are constructions for $i\mathcal{O}$ for RAM programs proposed where the size of the obfuscated program is independent of the running time [6, 11, 36]. However, by definition, these constructions do not achieve VBB obfuscation.

In order to circumvent the impossibility of VBB obfuscation, Goyal *et al.* [27] considered virtual black-box obfuscators on minimal secure hardware tokens. Goyal *et al.* show how to achieve VBB obfuscation for all polynomial time computable functions using *stateless* secure hardware tokens that only perform authenticated encryption/decryption and a single NAND operation. In a related line of work, Döttling *et al.* [17] show a construction for program obfuscation using a single stateless hardware token in universally input-oblivious models of computation. Bitansky *et al.* [5] show a construction for program obfuscation from "leaky" hardware. Similarly, Chung *et al.* [15] considered basing the closely related primitive of functional encryption on hardware tokens. Unfortunately, all the above works require the obfuscated program run using a universal circuit (or similar model) to achieve function privacy. They do not support running RAM programs directly. This severely limits the practicality of the above schemes, as we demonstrate in Section 6.5.

Oblivious RAMs. To enable running RAM programs directly on secure hardware, we use a hardware implementation of Oblivious RAM (ORAM) to hide access patterns to external memory. ORAM was introduced by Goldreich and Ostrovsky where they explored the use of tamper-proof hardware for software protection [26]. Recently, there has been a lot of work in making ORAMs practical. In this paper, we use an efficient hardware implementation of Path ORAM [52] called Tiny ORAM [20, 19].

Secure processors. Secure processors such as AEGIS [53], XOM [38], Bastion [13] and Intel SGX [42] encrypt and verify the integrity of main memory. Applications such as VC3 [48] that are built atop Intel SGX can run MapReduce computations [16] in a distributed cloud setting while keeping code and data encrypted. However, these secure processors do not hide memory access patterns. An adversary observing communication patterns between a processor and its memory can still infer significant information about the data [43, 57].

There have been some recent secure processor proposals that do hide memory access patterns [18, 41, 40, 45]. Ascend [18] is a secure processor architecture that protects privacy of data against physical attacks when running arbitrary programs. Phantom [41] similarly achieves memory obliviousness, and has been integrated with GhostRider [40] to perform program analysis and decide whether to use an encrypted RAM or Oblivious RAM for different memory regions. They also employ a scratchpad wherever applicable. Raccoon [45] hides data access patterns on commodity processors by evaluating all program paths and using an Oblivious RAM in software.

The primary difference between the above schemes and HOP is the following. All of the above schemes focused on protecting input data, while the program is assumed to be public and known to the adversary. GhostRider [40] even utilizes public knowledge of program behavior to improve performance through static analysis. Conversely, obfuscation and HOP protect the program and the input data is controlled by the adversary. We remark, however, that HOP can be extended to *additionally* achieve data privacy simply by adding routines to decrypt the (now private) inputs and encrypt the final outputs before they are sent to the client (now different from the HOP processor owner). Naturally, the enhanced security comes with additional cost. We evaluate this overhead of additionally providing program-privacy by comparing to GhostRider in Section 6.5.

Secure computation. There is a line of work addressing how to build a general purpose MIPS processor for garbled circuits [51, 56]. When one party provides the program, the system is capable of performing private function secure function evaluation (PF-SFE). Similarly, universal circuits [55, 35, 33] in combination with garbled circuits (which can be evaluated efficiently with techniques in [30]) or other multiparty computation protocols can be used to hide program functionality from one of the parties. The work of Katz [32] relies on trusted hardware tokens to circumvent the theoretical impossibility of UC-secure multi-party computation under dishonest majority. However, all the above results are in the context of secure computation, which is inherently interactive and only allows one-time use − i.e. for every input, both parties are involved in the computation. On the contrary, obfuscation requires that a party non-interactively execute the obfuscated program several times on multiple inputs.

Heuristic approaches to obfuscation. There are heuristic approaches to code obfuscation for resistance to reverse engineering [57, 31, 47]. These works provide low overheads, but do not offer any cryptographic security.

Terminology: Hardware Tokens. Trusted hardware is widely referred to as hardware *tokens* in the theoretical literature [32, 27, 17, 15]. Secure tokens are typically assumed to be minimal trusted hardware that support limited operations (e.g., a NAND gate in [27]). However, running programs in practice requires full-fledged processors. In this paper, we refer to HOP as "secure hardware" or a "secure processor". As a processor, HOP will store a lot more internal state (e.g., a register file, etc.). We note that from a theoretic perspective, both HOP and 'simple' hardware tokens require a number of gates which is polylogarithmic in memory size.

Terminology: Stateful vs. Stateless tokens. The literature further classifies secure tokens as either stateful tokens or stateless. A stateful token maintains state across invocations. On the other hand, a stateless token, except for a secret key, does not maintain any state across invocations. While HOP maintains state across most invocations for better performance, we will augment HOP to support on-demand context switching — giving the receiver the ability to swap out an obfuscated program for another at any time (Section 3.5), which is common in today's systems. In an extreme scenario, the adversary can context switch after every processor cycle. In this case, HOP becomes equivalent to a "stateless" token from a theoretical perspective [27, 15], and our security proof will assume stateless tokens.

3 Obfuscation from Trusted Hardware

In this section, we describe the HOP architecture. We will start with an overview of a simple (not practical) HOP processor to introduce some key points. Each subsection after that introduces additional optimizations (some expose security issues, which we address) to make the scheme more practical. We give security intuition where applicable, and formally prove security for the fully optimized scheme in Section 4.

3.1 Execution On-Chip

Let us start with the simplest case where the whole obfuscated program and its data (working set) fit in a processor's on-chip storage. Then, we may architect a HOP processor to be able to run programs whose working sets don't exceed a given size. In the setup phase, first, the sender correctly determines a value T – the amount of time (in processor cycles) that the program, given any input, runs on HOP. Then, the sender encrypts (obfuscates) the program together using an authenticated encryption scheme. T is authenticated along and included with the program but is public. The obfuscated program is sent to the receiver. The receiver then sends the obfuscated program and her own input to the HOP processor. The HOP processor decrypts and runs the program, and returns a result after T processor cycles. The HOP processor makes no external memory requests during its execution since the program and data fit on chip. Security follows trivially.

3.2 Adding External Memory

Unfortunately, since on-chip storage is scarce (commercial processors have a few MegaBytes of on-chip storage), the above solution can only run programs with small working sets. To handle this, like any other modern processor, the HOP processor needs to access an external memory, which is possibly controlled by the malicious receiver.

When the HOP processor needs to make an access to this receiver memory, it needs to hide its access patterns. For the purposes of this discussion, the access pattern indicates the processor's memory operations (reads vs. writes), the memory addresses for each access and the data read/written in each access. We hide access pattern by using an Oblivious RAM (ORAM), which makes a polylogarithmic number of physical memory accesses to serve each logical memory request from the processor [52]. The ORAM appears to HOP as an on-chip memory controller that intercepts memory requests from the HOP processor to the external memory. That is, the ORAM is a hardware block on the processor and is trusted. (More formal definitions for ORAM are given in Section 4.1.)

Each ORAM access can take thousands of processor cycles [19]. Executing instructions – once data is present on-chip – is still as fast as an insecure machine (e.g., several cycles). To hide when ORAM accesses are actually needed, HOP must make accesses at a static program-independent frequency (more detail below). As before, HOP runs for T time on all inputs and hence achieves the same privacy as the scheme in Section 3.1.

Generating T and security requirements. When accessing receiver-controlled memory, we must change T to represent some amount of work that is independent of the external memory's latency. That is, if T is given in processor cycles, the adversary can learn the true program termination time by running the program multiple times and varying the ORAM access latency each time (causing a different number of logical instructions to complete each time). To prevent this, we change T to mean 'the number of external memory read/writes made with the receiver.'

Integrity. To ensure authenticity of the encrypted program instructions and data during the execution, HOP uses a standard Merkle tree (or one that is integrated with the ORAM [46]) and stores the root of a Merkle tree internally. The receiver cannot tamper with or rewind the memory without breaking the Merkle tree authentication scheme.

Efficiency. While the above scheme can handle programs with large working sets, it is very inefficient. The problem is that each instruction may trigger multiple ORAM accesses. To give off the impression of running any program, we must provision for this worst case: running each instruction must incur the cost of the worst-case number of ORAM accesses. This can result in $\sim 10,000\times$ slowdown over an insecure processor.[3]

[3]Our ORAM latency from Section 6 is 3000 cycles. The RISC-V ISA [12] we adopt can trigger 3 ORAM accesses, one to fetch the instruction, 1 or 2 more to fetch the operand, depending on whether the operand straddles an ORAM block boundary.

The next two subsections discuss two techniques to securely reduce this overhead by over two orders of magnitude. These ideas are based on well-known observations that many programs have more arithmetic instructions than memory instructions, and exhibit locality in memory accesses.

3.3 Adding Instruction Scheduling

The key intuition behind our first technique is that many programs execute multiple arithmetic instructions for every memory access. For example, an instruction trace may be the following: 'A A A A M A A M', where A, M refer to arithmetic and memory instructions respectively.

Our optimization is to let the HOP processor follow a fixed and pre-defined schedule: N arithmetic instructions followed by one memory access. In the above example, given a schedule of A^4M, the processor would insert two *dummy* arithmetic instructions to adhere to the schedule. A dummy arithmetic instruction can be implemented by executing a `nop` instruction. The access trace observable to the adversary would then be:

A A A A M A A **A** **A** M

The bold face **A** letters refer to dummy arithmetic instructions introduced by the processor.

Likewise, if another part of the program trace contains a long sequence of arithmetic instructions, the processor will insert dummy ORAM accesses to adhere to the schedule.

Gains. For most programs in practice, there exists a schedule with $N > 1$ that would perform better than our baseline scheme from Section 3.2. For $(N + 1)$ instructions, the baseline scheme performs $(N + 1)$ arithmetic and memory accesses. With an $A^N M$ schedule, our optimized scheme performs only one memory access which translates to a speedup of $N\times$ in the best case, when the cost of the memory access is much higher than an arithmetic instruction. To translate this into performance on HOP - given that HOP must run for T time - consider the following: If $N > 1$ does improve performance for the given program on all inputs, it means the sender can specify a smaller T for that program, while still having the guarantee that the program will complete given any input. A smaller T means better performance.

Setting N and security intuition. We design all HOP processors to use the same value of N for all programs and all inputs (i.e., N is set at HOP manufacturing time like the private key). More concretely, we set

$$N = \frac{\text{ORAM latency}}{\text{Arithmetic latency}}$$

In other words, the number of processor cycles spent on arithmetic instructions and memory instructions are the same. For typical parameter settings, $N > 1000$ is expected. While this may sound like it will severely hurt performance given pathological programs, we show that this simple strategy does "well" on arbitrary programs and data, formalized below.

Claim: For *any* program and input, the above N results in $\leq 50\%$ of processor cycles performing dummy work.

We refer the reader to the full version of this paper for a proof of this claim. The claim implies that in comparison to a solution that *does not* protect the main memory timing channel, our fixed schedule introduces a maximum overhead of $2\times$ given any program − whether they are memory or computation intensive. Said another way, even when more sophisticated heuristics than a fixed schedule are used for different applications, the performance gain from those techniques is a factor of 2 at most.

Security. We note that our instruction scheduling scheme does not impact security because we use a fixed, public N for all programs.

3.4 Adding on-chip Scratchpad Memory

Our second optimization adds a scratchpad: a *small* unit of trusted memory (RAM) inside the processor, accesses to which are not observable by the adversary.[4] It is used to temporarily store a portion of the

[4] We remark that we use a software-managed scratchpad (as opposed to a conventional processor cache) as it is easier to determine T when using a scratchpad.

working set for programs that exhibit locality in their access patterns.

Running programs with a scratchpad. We briefly cover how to run programs using a scratchpad here. More (implementation-specific) detail is given in Section 5.1. At a high level, data is loaded into the scratchpad from ORAM/unloaded to ORAM using special (new) CPU instructions that are added to the obfuscated program. These instructions statically determine when to load which data to specified offsets in the scratchpad. Now, the scratchpad load/unload instructions are the only instructions that access ORAM (i.e., are the only 'M' instructions). Memory instructions in the original program (e.g., normal loads and stores) merely lookup the scratchpad inside the processor (these are now considered 'A' instructions). We will assume the program is correctly compiled so that whenever a program memory instruction looks up the scratchpad, the data in question has been put there sometime prior by a scratchpad load/unload instruction.

Security intuition. When the program accesses the scratchpad, it is hidden from the adversary since this is done on-chip. As before, the only adversary-visible behavior is when ORAM is accessed and this will be governed by the program-independent schedule from Section 3.3.

Program independence. We note that HOP with a scratchpad is still program independent. Multiple programs can be written (and obfuscated) for the same HOP processor. One minor limitation, however, is that once an obfuscated program is compiled, it must be compiled with 'minimum scratchpad size' specified as a new parameter and cannot be run on HOP processors that have a smaller scratchpad. This is necessary because having a smaller scratchpad will increase T by some unknown amount. If the program is run on a HOP processor with a larger scratchpad, it will still function but some scratchpad space won't be used.

Gains. In the absence of a scratchpad, the ratio of arithmetic to memory instructions is on average 5:1 for our workloads. When using a scratchpad, a larger amount of data is stored by the processor, thus decreasing memory accesses. This effectively decreases the execution time T of the program and substantially improves performance for programs with high locality (evaluated in Section 6.3).

3.5 Adding context switching and stateless tokens

A problem with the proposals discussed so far is that once a program is started, it cannot be stopped until it returns a response. But a user may wish to concurrently run multiple obfuscated programs for a practical deployment model. Therefore, we design the HOP processor to support on-demand context switch, i.e., the receiver can invoke a context switch at any point during execution. This, however, introduces security problems that we need to address.

A context switch means that the current program state should be swapped out from the HOP processor and replaced with another program's state. Since such a context switch can potentially happen at every invocation, the HOP processor no longer stores state and is a stateless token. In such a scenario, we design it to encrypt all its internal state, and send this encrypted/authenticated state (denoted $\overline{\mathsf{state}}$) to the receiver (i.e., the adversary) on a context switch. Whenever the receiver passes control back to the token, it will pass back the encrypted state as well, such that the token can "recover" its state upon every invocation.

Challenges. Although on the surface, this idea sounds easy to implement, in reality it introduces avenues for new attacks that we now need to defend against. For the rest of the paper, and in-line with real processors, we assume the only data that remains in HOP is the per-chip secret key (Section 1). A notable attack is the *rewinding* attack. In this attack, instead of passing to the token the correct and fresh encrypted $\overline{\mathsf{state}}$ as well as fresh values of memory reads, a malicious receiver can pass old values.[5] The receiver can also *mix-and-match* values from entirely different executions of the same program or different programs. The rest of the section outlines how to prevent the above attacks. We remark that while the below have simple fixes,

[5]Here is a possible attack by which the adversary can distinguish between two access patterns. Consider the access pattern $\{a, a\}$ i.e., accessing the same block consecutively. If a tree-based ORAM [49] is used, after the first access, the block is remapped to a new path l' and the new path l' would be subsequently accessed. If the adversary rewinds and executes again, the block may be mapped to a different path l''. Thus, for two different executions, two different paths (l' and l'') are accessed for the second access. Note that for another access pattern $\{a, b\}$ for $a \neq b$, the same paths would be accessed even after rewinding, thus enabling the adversary to distinguish between access patterns.

the problems themselves are easy to overlook and underscore the need for a careful formal analysis. Indeed, we discovered several of these issues while working through the security proof itself.

Preventing mix-and-match. To prevent this attack, we enforce that the receiver must submit an encrypted state $\overline{\mathsf{state}}$, corresponding to an execution at some point t, along with a matching read from time t for the same execution. To achieve this, observe that $\overline{\mathsf{state}}$ is encrypted with a IND-CPA + INT-CTXT-secure authenticated encryption scheme, and that the state carries all necessary information to authenticate the next memory read. The state contains information unique to the specific program, the specific program execution, and to the specific instruction that the token expects.

Preventing rewinding during program execution. An adversary may try to gain more information by rewinding an execution to a previous time step, and replaying it from that point on. To prevent an adversary from learning more information in this way, we make sure that the token simply replays an old answer should rewinding happen — this way, the adversary gains no more information by rewinding. To achieve this, we make sure that any execution for a (program, inp) pair is entirely deterministic no matter how many times you replay it. All randomness required by the token (e.g., those required by the ORAM or memory checker) are generated pseudorandomly based on the tuple (K, H_S, H_R) where K is a secret key hardwired in the token, H_R is a commitment to the receiver's input and $H_S := \mathsf{digest}(\overline{\mathsf{mem}_0})$ is a Merkle root of the program.

Preventing rewinding during input insertion. In our setting, the obfuscated program's inputs inp are chosen by the receiver. Since inputs can be long, it may not be possible to submit the entire input in one shot. As a result, the receiver has to submit the input word by word. Therefore the malicious receiver may rewind to a point in the middle of the input submission, and change parts of the input in the second execution. Such a rewinding causes two inputs to use the same randomness for some part of the execution.

To prevent such an input rewinding attack, we require that the adversary submit a Merkle tree commitment $H_R := \mathsf{digest}(\mathsf{inp})$ of its input inp upfront, before submitting a long input word by word. H_R uniquely determines the rest of the execution, such that any rewinding will effectively cause the token to play old answers (as mentioned above), and the adversary learns nothing new through rewinding.

4 Formal Scheme

We now give a formal model for the fully optimized HOP processor (i.e., including all subsections in Section 3) and prove its security in UC framework. Section 4.1 describes the preliminaries. Section 4.2 describes the ideal functionality for obfuscation of RAM programs. Sections 4.3 and 4.4 describe our formal scheme and proof in the UC framework.

4.1 Preliminaries

The notations used in this section are summarized in Table 1. We denote the assignment-operator with :=, while we use = to denote equality. Encryption of data is denoted by an overline, e.g., $\overline{\mathsf{state}} = \mathsf{Enc}_K(\mathsf{state})$, where Enc denotes a IND-CPA + INT-CTXT-secure authenticated encryption scheme and K is the key used for encryption.

Universal Composability framework. The Universal Composability framework [10] considers two worlds – 1. *real world* where the parties execute a protocol π. An adversary $\mathcal{A}$ controls the corrupted parties. 2. *ideal world* where we assume the presence of a trusted third party. The parties interact with a trusted third party (also called ideal functionality $\mathcal{F}$) with a protocol ϕ. A simulator $\mathcal{S}$ tries to mimic the actions of $\mathcal{A}$. Intuitively, the amount of information revealed by π in the real world should not be more than what is revealed by interacting with the trusted third party in the ideal world. In other words, we have the following: an environment $\mathcal{E}$ observes one of the two worlds and guesses the world. Protocol π UC-realizes ideal functionality $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$, such that an environment $\mathcal{E}$ cannot distinguish (except with negligible probability) whether it is interacting with $\mathcal{S}$ and ϕ or with $\mathcal{A}$ and π.

Table 1: Notations

K	Hardwired secret key stored by the token
mem_0	A program as a list of instructions
inp	Input to the program
mem	Memory required for program execution
outp	Program output
$\ell_{\mathsf{in}}, \ell_{\mathsf{out}}, w$	Bit-lengths of input, output, and memory word
N	Number of words in memory
T	Time for program execution
$\mathsf{RAM.params}$	$\{T, N, \ell_{\mathsf{in}}, \ell_{\mathsf{out}}, w\}$
$\mathsf{oramstate}$	State stored by ORAM
$\mathsf{sstorestate}$	State stored by sstore
H_R	Digest of receiver's input, i.e., $\mathsf{digest}(\mathsf{inp})$
H_S	Digest of sender's program, i.e., $\mathsf{digest}(\overline{\mathsf{mem}_0})$
H'	Merkle root of the main memory

Random Access Machines. We now give definitions for Random Access Machine (RAM) programs, a basic processor model for RAM programs. Let $\mathcal{RAM}[\Pi, T, N, \ell_{\mathsf{in}}, \ell_{\mathsf{out}}, w]$ denote a family of RAM programs with the following public parameters: Π denotes the next instruction circuit; T denotes the number of steps the program will be executed; w denote the bit-width of a memory word; and N, ℓ_{in} and ℓ_{out} denote the memory, input and output lengths respectively (in terms of number of words).

We consider programs $\mathsf{RAM} := \langle \mathsf{cpustate}, \mathsf{mem} \rangle \in \mathcal{RAM}[\Pi, T, N, \ell_{\mathsf{in}}, \ell_{\mathsf{out}}, w]$ to be a tuple, where $\mathsf{cpustate}$ denotes the CPU's initial internal state, and mem denotes an initial memory array. In these programs, for each step of the execution, the next instruction function is executed over the old $\mathsf{cpustate}$ and the most recently fetched w bit memory word denoted rdata:

$$(\mathsf{cpustate}, \mathsf{op}) := \Pi(\mathsf{cpustate}, \mathsf{rdata})$$

As a result, $\mathsf{cpustate}$ is updated, and a next read/write instruction op is fetched. Initially, rdata is set to 0.

On input inp, the execution of $\mathsf{RAM}[T, N, \ell_{\mathsf{in}}, \ell_{\mathsf{out}}, w] := \langle \Pi, \mathsf{cpustate}, \mathsf{mem} \rangle$ is defined as the following:

```
$\mathsf{rdata} := 0$
$\mathsf{mem}[1..\ell_{\mathsf{in}}] := \mathsf{inp}$
for $t \in [1, 2, \ldots, T]$:
  $(\mathsf{cpustate}, \mathsf{op}) := \Pi(\mathsf{cpustate}, \mathsf{rdata})$
  if $\mathsf{op} = (\mathtt{write}, addr, \mathsf{wdata})$
    $\mathsf{mem}[addr] := \mathsf{wdata}$
  else if $\mathsf{op} = (\mathtt{read}, addr, \perp)$
    $\mathsf{rdata} := \mathsf{mem}[addr]$
Output $\mathsf{rdata}$    // $\mathsf{rdata}$ stores the output
```

For notational simplicity, we assume that output length ℓ_{out} is small and can be stored in rdata. However, our results can be extended easily to larger values of ℓ_{out}. For succinctness, we denote $(T, N, \ell_{\mathsf{in}}, \ell_{\mathsf{out}}, w)$ by $\mathsf{RAM.params}$. Wherever its clear from context, we abuse notation to denote $\mathcal{RAM}[\Pi, T, N, \ell_{\mathsf{in}}, \ell_{\mathsf{out}}, w]$ as $\mathcal{RAM}$.

Oblivious RAM. Let mem denote a memory array that supports two types of operations: a) On $(\mathtt{read}, addr)$, it outputs $\mathsf{mem}[addr]$; b) On $(\mathtt{write}, addr, \mathsf{wdata})$, it sets $\mathsf{mem}[addr] := \mathsf{wdata}$, and outputs $\perp$. In this paper, we define an Oblivious RAM as a *stateful*, probabilistic algorithm that interacts with a memory array mem. It is denoted as $\mathsf{ORAM}^{N,w}$ where N and w are public parameters denoting the memory capacity in terms of number of words, and the bit-width of a word. mem denotes the initial state of the memory, where all but the first N locations are set to 0. An ORAM converts memory contents mem to mem'. An ORAM takes two

```
$\mathcal{F}_{obf}^{\mathcal{RAM}}$[sender, receiver ]

On receive (“create”, $\mathsf{RAM}$) from sender for the first time:
    Create a unique nonce denoted $\mathsf{pid}$
    Store $(\mathsf{pid}, \mathsf{RAM})$, send (“create”, $\mathsf{pid}$) to receiver

On receive (“execute”, $\mathsf{pid}, \mathsf{inp}$) from receiver:
    assert $(\mathsf{pid}, \mathsf{RAM})$ is stored for some $\mathsf{RAM}$
    $\mathsf{outp} := \mathsf{RAM}(\mathsf{inp})$, send $\mathsf{outp}$ to receiver
```

Figure 2: **Ideal Functionality $\mathcal{F}_{obf}^{\mathcal{RAM}}$.** Although there can be multiple instances of this ideal functionality, we omit writing the session identifier explicity without risk of ambiguity. In this paper, we adopt the same UC notational conventions as Pass, Shi, and Tramer [44]. In particular, we parametrize each functionality and protocol by its session identifier, and the identifiers of the parties involved — although in this paper, we omit writing the session identifier explicitly without risk of ambiguity.

types of inputs: $\mathsf{op} := (\mathtt{read}, addr)$, and $\mathsf{op} := (\mathtt{write}, addr, \mathsf{wdata})$. After receiving input op_i, ORAM interacts with mem', and produces read/write operations into mem' as output, denoted by $\vec{\mathsf{op}}_i$. These operations $\vec{\mathsf{op}}_i$ implicitly define memory contents of mem.

We say that an ORAM algorithm is correct, if for any n, for any input sequence $(\mathsf{op}_1, \ldots, \mathsf{op}_n)$, ORAM outputs correctly. In other words, the memory contents of mem implicitly defined by mem' after execution of $(\vec{\mathsf{op}}_1, \ldots, \vec{\mathsf{op}}_n)$ is identical to the memory contents of mem defined by executing $(\mathsf{op}_1, \ldots, \mathsf{op}_n)$ on mem. We say that an ORAM scheme ORAM is *oblivious* if there exists a polynomial-time simulator Sim such that no polynomial time adversary $\mathcal{A}$ can distinguish between the transcript of the real ORAM execution and a simulated transcript that Sim outputs. Sim is given only N and w, even when the simulated memory access are provided one-by-one to $\mathcal{A}$.

Remark: ORAM initialization. In this paper, we assume an ORAM starts out with a memory array where the first N words are non-zero (reflecting the initial unshuffled memory), followed by all zeros. Most ORAM schemes require an initialization procedure to shuffle the initial memory contents. In this paper, we assume that the ORAM algorithm performs a linear scan of first N memory locations and inserts them into ORAM. This is used by the simulator in our proof to extract the input used for execution of the program. We use the convention that such initialization is performed by the ORAM algorithm upon the first read or write operation — therefore our notation does not make such initialization explicit. This also means that the first ORAM operation will incur a higher overhead than others.

4.2 $\mathcal{F}_{obf}^{\mathcal{RAM}}$: Modeling Obfuscation in UC

The ideal functionality for obfuscation $\mathcal{F}_{obf}^{\mathcal{RAM}}$ is described in Figure 2. The sender sends the description of a RAM program, $\mathsf{RAM} \in \mathcal{RAM}$ and a program ID pid, using the “create” query. The functionality stores this program, pid, the sender and receiver. When the receiver invokes “execute” query on an input inp, it evaluates the program on inp, and returns output outp.

4.3 Scheme Description

We now provide the complete description of our scheme. We model the secure hardware token through the $\mathcal{F}_{token}$ functionality (Figure 3). Our construction realizes $\mathcal{F}_{obf}^{\mathcal{RAM}}$ in the $\mathcal{F}_{token}$-hybrid model [27] and is described in Figure 4.

In order to account for all possible token queries that may be required for an ORAM scheme, $\mathcal{F}_{token}$ relies on an internal, transient instance of $\mathcal{F}_{internal}$ to execute each step of the program evaluation. Each time $\mathcal{F}_{token}$ yields control to the receiver, the entire state of $\mathcal{F}_{internal}$ is destroyed. Whenever the receiver calls back $\mathcal{F}_{token}$ with $\overline{\mathsf{state}}$, $\mathcal{F}_{token}$ once again creates a new, transient instance of $\mathcal{F}_{internal}$, sets its state to the decrypted state, and invokes $\mathcal{F}_{internal}$ to execute next step.

The sender. Let the program to be obfuscated be $\mathsf{RAM} := \langle \mathsf{cpustate}_0, \mathsf{mem}_0 \rangle$ where mem_0 is a list of program instructions. The sender first creates the token containing a hardwired secret key K where $K := (K_1, K_2, K_3)$. K_1 is used as the encryption key for encrypting state, K_2 is used as the key to a pseudorandom function used by the ORAM and K_3 is used as the key for a pseudorandom function used by sstore (described later). This is modeled by our functionality using the "store key" query (Figure 4 line 1). The sender then encrypts mem_0 (one instruction at a time) to obtain $\overline{\mathsf{mem}_0}$. It creates a Merkle root $H_S := \mathsf{digest}(\overline{\mathsf{mem}_0})$, which is used by $\mathcal{F}_{token}$ during execution to verify integrity of the program. The sender creates an encrypted header $\overline{\mathsf{header}} := \mathsf{Enc}_{K_1}(\mathsf{cpustate}_0, H_S, \mathsf{RAM.params})$ where $\mathsf{RAM.params} = \{T, N, \ell_{\mathsf{in}}, \ell_{\mathsf{out}}, w\}$. The sender sends $\overline{\mathsf{header}}$, $\overline{\mathsf{mem}_0}$, and $\mathsf{RAM.params}$ as the obfuscated program to the receiver. As the obfuscated program consists of only the encrypted program and metadata, for a program of size P bits, the obfuscated program has size $P + O(1)$ bits. In the real world, the sender sends the hardware token with the functionality $\mathcal{F}_{token}$ to the receiver. The receiver can use the same stateless token to execute multiple obfuscated programs sent by the sender.

The receiver. On the receiver's side, the token functionality makes use of an ORAM and a secure store sstore. The token functionality (trusted hardware functionality) is modeled by an augmented RAM machine.

1. **ORAM.** ORAM takes in $[\kappa := \mathsf{PRF}_{K_2}(\mathsf{ssid}), \mathsf{oramstate}]$ (where $\mathsf{ssid} := (H_S, H_R)$) as internal secret state of the algorithm. κ is a session-specific seed used to generate all pseudorandom numbers needed by the ORAM algorithm — recall that all randomness needed by ORAM is replaced by pseudorandomness to avoid rewinding attacks. As mentioned in Section 4.1, we assume that the ORAM initialization is performed during the first read/write operation. At this point, the ORAM reads the first N memory locations to read the program and the input, and inserts them into the ORAM data structure within mem.

2. **Secure store module sstore.** sstore is a stateful deterministic secure storage module that sits in between the ORAM module and the untrusted memory implemented by the receiver. Its job is to provide appropriate memory encryption and authentication. sstore's internal state includes $\kappa := \mathsf{PRF}_{K_3}(\mathsf{ssid})$ and $\mathsf{sstorestate}$. $\mathsf{sstorestate}$ contains a succinct digest of program, input and memory to perform memory authentication. κ is a session-specific seed used to generate all pseudorandom numbers for memory encryption.
   At the beginning of an execution, $\mathsf{sstorestate}$ is initialized to $\mathsf{sstorestate} := (H_S, H_R, H' := 0)$, where H_S denotes the Merkle root of the encrypted program provided by the sender, H_R denotes the Merkle root of the (cleartext) input and H' denotes the Merkle root of the memory mem. By convention, we assume that if a Merkle tree or any subtree's hash is 0, then the entire subtree must be 0. The operational semantics of sstore is as follows: upon every data access request $(\mathtt{read}, addr)$ or $(\mathtt{write}, addr, \mathsf{wdata})$:
   - If $addr$ is in the mem_0 part of the memory (the sender-provided encrypted program), interact with mem and use H_S to verify responses. Update H_S appropriately if the request type is $\mathtt{write}$.
   - If $addr$ is in the inp part of the memory (the receiver-provided input), interact with mem and use H_R to verify responses.
   - Otherwise, interact with mem and use H' to verify responses. Update H' appropriately.

   Upon successful completion, sstore outputs the data fetched for read requests, and outputs 0 or 1 for write requests. Note that the sstore algorithm simply aborts if any of the responses fail verification.

3. **Augmented Random Access Machines.** We now extend the RAM model to support instruction scheduling and a scratchpad (Sections 3.3 and 3.4). $\mathcal{RAM}$ can be augmented to use a next instruction circuit $\Pi' := \Pi^N$ for a fixed N, with the following modifications:
   (a) Π' is a combinational circuit, which consists of N next-instruction circuits Π_i cascaded as shown in Figure 5.

```
                         $\mathcal{F}_{token}$ [sender, receiver ]
// Store the secret key $K$ in the token
On receive ("store key", $K$) from sender:
  Store the secret key $K$, ignore future "store key" inputs
  Send "done" to sender

// This step commits the receiver to his input through $H_R$
On receive ("initialize", $\overline{\mathsf{header}}$, $H_R$) from receiver:
  Parse $K := (K_1, K_2, K_3)$
  $(\mathsf{cpustate}_0, H_S, \mathsf{RAM.params}) := \mathsf{Dec}_{K_1}(\overline{\mathsf{header}})$; abort if fail
  $\mathsf{state} := \{\mathsf{ssid} := (H_S, H_R),\ time := 0,$
           $\mathsf{rdata} := 0,\ \mathsf{cpustate} := \mathsf{cpustate}_0,$
           $\mathsf{sstorestate} := (\text{"init"}, H_S, H_R, H' := 0),$
           $\mathsf{oramstate} := \text{"init"}, \mathsf{params} := \mathsf{RAM.params}\}$
  send $\overline{\mathsf{state}} := \mathsf{Enc}_{K_1}(\mathsf{state})$ to receiver

On receive (_) from $\mathcal{F}_{internal}$: // ORAM queries
  $\overline{\mathsf{state}} := \mathsf{Enc}_{K_1}(\mathcal{F}_{internal}.\mathsf{state})$
  send (_, $\overline{\mathsf{state}}$) to receiver

On receive (_, $\overline{\mathsf{state}}$) from receiver: // ORAM queries
  $\mathsf{state} := \mathsf{Dec}_{K_1}(\overline{\mathsf{state}})$, abort if fail
  Instantiate a new instance $\mathcal{F}_{internal}$, set $\mathcal{F}_{internal}.\mathsf{state} := \mathsf{state}$, and $\mathcal{F}_{internal}.K := K$
  Send _ to $\mathcal{F}_{internal}$

                         $\mathcal{F}_{internal}$
  Define $\mathcal{F}_{internal}.\mathsf{state} \overset{\text{alias}}{:=} (\mathsf{ssid}, time, \mathsf{rdata}, \mathsf{cpustate}, \mathsf{sstorestate}, \mathsf{oramstate}, \mathsf{params})$

  // execute program
  On receive ("execute one step") from $\mathcal{F}_{token}$:
   1: assert $time \leq \mathsf{params}.T$
   2: $(\mathsf{cpustate}, \mathsf{op}) \leftarrow \Pi'(\mathsf{cpustate}, \mathsf{rdata})$
   3: Send op to $\mathsf{ORAM}[\mathsf{PRF}_{K_2}(\mathsf{ssid}), \mathsf{oramstate}] \Leftrightarrow \mathsf{sstore}[\mathsf{PRF}_{K_3}(\mathsf{ssid}), \mathsf{sstorestate}] \Leftrightarrow \mathcal{F}_{token}$, wait for
      output from ORAM, abort if sstore aborts; /* instantiate ORAM with state oramstate, instantiate
      sstore with state sstorestate, connect ORAM's communication tape to sstore's input tape, connect
      sstore's communication tape to caller $\mathcal{F}_{token}$. This represents a multi-round protocol. */
   4: If $\mathsf{op} = (\mathtt{read}, \ldots)$, let $\mathsf{rdata} :=$ output
   5: $time := time + 1$
   6: If $time = \mathsf{params}.T$: send ("okay", rdata) to $\mathcal{F}_{token}$ ; else send ("okay", $\perp$) to $\mathcal{F}_{token}$
```

Figure 3: **Functionality $\mathcal{F}_{token}$.** For succinctness, encryption of some data is represented using an overline on it, e.g., $\overline{\mathsf{state}} = \mathsf{Enc}_{K_1}(\mathsf{state})$, where Enc denotes a IND-CPA + INT-CTXT-secure authenticated encryption scheme. "_" denotes a wildcard field that matches any string.

```
$\mathbf{Prot}_{obf}$[sender, receiver ]
Sender:
On receive ("create", $\mathsf{RAM} = \langle \mathsf{cpustate}_0, \mathsf{mem}_0 \rangle$) from env:
 1: If not initialized: $K := (K_1, K_2, K_3) \xleftarrow{\$} \{0,1\}^{3\lambda}$, send ("store key", $K$) to $\mathcal{F}_{token}$, await "done"
 2: $\overline{\mathsf{mem}}_0 := \{\mathsf{Enc}_{K_1}(\mathsf{mem}_0[i], \mathsf{rand}())\}_{i \in |\mathsf{mem}_0|}$
 3: $H_S := \mathsf{digest}(\overline{\mathsf{mem}}_0)$    // $H_S$: program Merkle root
 4: $\overline{\mathsf{header}} := \mathsf{Enc}_{K_1}(\mathsf{cpustate}_0 || H_S || \mathsf{RAM.params}, \mathsf{rand}())$
 5: Send $(\overline{\mathsf{header}}, \overline{\mathsf{mem}}_0, \mathsf{RAM.params})$ to receiver

Receiver:
On receive ("execute", $\mathsf{pid}, \mathsf{inp}$) from env:
 1: Await $(\overline{\mathsf{header}}, \overline{\mathsf{mem}}_0, \mathsf{RAM.params})$ from sender s.t. $\mathsf{RAM.params}.H_S = \mathsf{pid}$ if not received already
 2: Initialize $\mathsf{mem} := \overline{\mathsf{mem}}_0 || \mathsf{inp} || \vec{0}$
 3: Send ("initialize", $\overline{\mathsf{header}}, H_R := \mathsf{digest}(\mathsf{inp})$) to $\mathcal{F}_{token}$, await $\overline{\mathsf{state}}$ from $\mathcal{F}_{token}$
 4: for $t$ in $\{1, \ldots, T\}$:
 5:    Send ("execute one step", $\overline{\mathsf{state}}$) to $\mathcal{F}_{token}$
 6:    Await $(\mathsf{oper}, \overline{\mathsf{state}})$ from $\mathcal{F}_{token}$;    // $\overline{\mathsf{state}}$ overwritten with the received value
 7:    Until $\mathsf{oper}$ = ("okay", _), repeat:    //multiple memory requests for the RAM step due to ORAM
 8:       perform the operation $\mathsf{oper}$ on $\mathsf{mem}$ and let the response be $\mathsf{res}$
 9:       forward $(\mathsf{res}, \overline{\mathsf{state}})$ to $\mathcal{F}_{token}$, and await $(\mathsf{oper}, \overline{\mathsf{state}})$ from $\mathcal{F}_{token}$;
10: Parse $\mathsf{oper}$ := ("okay", $\mathsf{outp}$), output $\mathsf{outp}$
```

Figure 4: **Protocol $\mathbf{Prot}_{obf}$.** Realizes $\mathcal{F}_{obf}^{\mathcal{RAM}}$ in the $\mathcal{F}_{token}$-hybrid model.

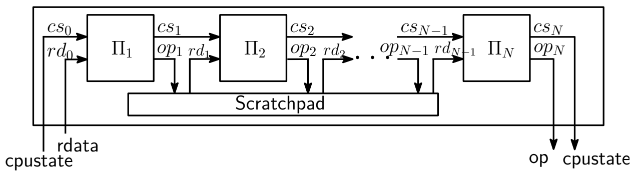

Figure 5: **Augmented Random Access Machine.** In this figure, $\mathsf{cpustate}_i$ is denoted by cs_i and rdata_i is denoted by rd_i.

(b) The Π_i's use an additional shared memory, referred to as $\mathsf{scratchpad}$. Each Π_i (except Π_1) operates on the output of Π_{i-1} and an operand rdata_{i-1} read from scratchpad. The next instruction circuit Π' outputs op_N to retrieve rdata from mem, which is subsequently used by Π_1.

On input inp, the execution of $\mathsf{RAM}[T, N, \ell_{\mathsf{in}}, \ell_{\mathsf{out}}, w] := \langle \Pi', \mathsf{cpustate}, \mathsf{mem} \rangle$ is similar to what was defined in Section 4.1 but uses Π' as the next instruction circuit. The augmented random access machine $\mathcal{RAM}'$ models a RAM that performs N instructions followed by an ORAM access. If some op_i cannot be served by the scratchpad, subsequent Π_j for $i + 1 \leq j \leq N$ do not update $\mathsf{cpustate}_j$ and output $\mathsf{op}_N = \mathsf{op}_i$ to load the required data in scratchpad.

Remark. For augmented random access machines that uses a scratchpad, rdata would typically be larger than a memory word (e.g. 512 bits).

We now explain how the receiver executes the program using the token described in Figure 3 and protocol in Figure 4.

Program execution. For ease of explanation, let us first assume that the ORAM is initialized and contains the program and input. The execution for any input proceeds in T time steps (Figure 4 line 4). At each time step, the receiver interacts with the token with two types of queries. For each type of query, $\mathcal{F}_{token}$ decrypts $\overline{\mathsf{state}}$ (aborts if decryption fails), instantiates $\mathcal{F}_{internal}$ with state and forwards the request to $\mathcal{F}_{internal}$. At the end of query, the $\overline{\mathsf{state}}$ is sent to the receiver along with the query response.

- **Execute one step**: This is shown in Figure 3 and Figure 4 line 5. When this query is invoked, $\mathcal{F}_{internal}$ executes the next instruction circuit Π' of the RAM machine to obtain an updated $\mathsf{cpustate}$ and an $\mathsf{op} \in \{\texttt{read, write}\}$. Once operation op is performed by the ORAM algorithm, $\mathcal{F}_{internal}$ updates $\mathsf{state}.time$ to reflect the execution of the instruction (Figure 3 line 5). The message "okay" is then sent to the receiver. At $time = T$, $\mathcal{F}_{internal}$ returns the program output to the receiver (Figure 3 line 6).
- **ORAM queries**: ORAMs can use a multi-round protocol (with possibly different types of queries) to read/write (Figure 3 line 3). It interacts with mem stored at the receiver through $\mathcal{F}_{token}$ (Figure 4 lines 7-9). To account for instantiation of any ORAM, $\mathcal{F}_{token}$ is shown to receive any query from receiver (indicated by wildcard (_) in Figures 3 and 4). These queries are sent to $\mathcal{F}_{internal}$ and vice-versa.

For each interaction with mem, sstore encrypts (resp. decrypts) data sent to (resp. from) the receiver. Moreover, sstore authenticates the data sent by the receiver. This completes the description of execution of the program.

Initialization. To initialize the execution, the receiver first starts by storing the program and input inp in its memory $\mathsf{mem} := \overline{\mathsf{mem}_0}||\mathsf{inp}||\vec{0}$. It commits to its input by invoking "initialize" (Figure 4 line 3) and sending a Merkle root of its input ($H_R = \mathsf{digest}(\mathsf{inp})$) along with $\overline{\mathsf{header}} := \mathsf{Enc}_{K_1}(\mathsf{cpustate}_0||H_S||\mathsf{RAM.params})$. $\mathcal{F}_{token}$ initializes the parameters, creates $\overline{\mathsf{state}}$ and sends it to the receiver.

The ORAM and sstore are initialized during the first invocation to "execute one step", i.e., $t = 1$ in Figure 4, line 4. The required randomness is generated pseudorandomly based on (K_2, H_S, H_R) for ORAM and (K_3, H_S, H_R) for sstore. As mentioned in Section 4.1, during initialization, ORAM in $\mathcal{F}_{token}$ reads mem_0 word by word (not shown in figure). For each word read, sstore performs Merkle tree verification with $H_s := \mathsf{digest}(\mathsf{mem}_0)$. Similarly, when the input is read, sstore verifies it with $H_R := \mathsf{digest}(\mathsf{inp})$. $\mathsf{sstorestate}$ and $\mathsf{oramstate}$ uniquely determine the initialization state. Hence, if the receiver rewinds, the execution trace remains the same. The commitment H_R ensures that the receiver cannot change his input after invoking "initialize". This completes the formal scheme description of the UC functionality $\mathcal{F}_{token}$.

4.4 Proof of Security

Theorem 1. *Assuming that Enc is an INT-CTXT + IND-CPA authenticated encryption scheme,* ORAM *satisfies obliviousness (Section 4.1),* sstore *adopts a semantically secure encryption scheme and a collision resistant Merkle hash tree scheme and the security of* PRF*, the protocol described in Figures 3 and 4 UC realizes* $\mathcal{F}_{obf}^{\mathcal{RAM}}$ *(Figure 2) in the* $\mathcal{F}_{token}$*-hybrid model.*

Description of the simulator. The ideal world simulator simulates the honest sender and $\mathcal{F}_{token}$.

- The simulator can receive the following three types of valid queries: "create" queries, "initialize" queries, and execution queries. An execution query is either of the format ("execute one step", $\overline{\mathsf{state}}$) or a response to a memory request. Henceforth we assume that all responses to memory requests are of the form ("mem", _, $\overline{\mathsf{state}}$).

- At the beginning, the simulator generates a random key K_1.

- Whenever the simulator receives ("create", pid) from $\mathcal{F}_{obf}^{\mathcal{RAM}}$, it sends $\overline{\mathsf{mem}_0} := \{\mathsf{Enc}_{K_1}(\vec{0})\}_{i \in |\mathsf{mem}_0|}$. $\overline{\mathsf{header}} := \mathsf{Enc}_{K_1}(\vec{0}||H_S := \mathsf{digest}(\overline{\mathsf{mem}_0})||\mathsf{RAM.params})$, and $\mathsf{RAM.params}$ to the receiver.

- Whenever the adversary (i.e., receiver) sends ("initialize", $\overline{\mathsf{header}}$, H_R): if the simulator has not sent the adversary $\overline{\mathsf{header}}$ before as a result of a "create" query, abort. At this time, a new subsession identified by $\mathsf{ssid} := (H_S, H_R)$ is created where H_S is contained in $\overline{\mathsf{header}}$, and $\mathsf{sstorestate}$ and $\mathsf{oramstate}$ for this subsession are initialized honestly.

  The simulator sends $\overline{\mathsf{state}} := \mathsf{Enc}_{K_1}(\vec{0})$ to the adversary.

- For each subsession identified by ssid, the simulator maintains the following:

- If $\overline{\mathsf{state}}$ was sent to the adversary during a subsession ssid, then the simulator remembers a tuple

$$(\mathsf{sstorestate}, \mathsf{oramstate})$$

which denotes the state of the execution when $\overline{\mathsf{state}}$ was sent to the adversary.

- The simulator forks a new ORAM simulator upon the creation of every new subsession, the term $\mathsf{oramstate}$ is the state of the (stateful) ORAM simulator.
- The simulator forks an honest instance of sstore upon the creation of every new subsession. The sstore instance is initialized with a randomly generated key, and the term $\mathsf{sstorestate}$ is its internal state.

- Whenever the simulator receives any execution query from the adversary, it checks if the $\overline{\mathsf{state}}$ received has been sent to the adversary before. If not, the simulator aborts. Else, continue with the following.
  - If the adversary has sent the same execution query before, the simulator replays the old answer from before with the following exception: for the $\overline{\mathsf{state}}$ contained in the answer, the simulator will re-encrypt $\overline{\mathsf{state}} := \mathsf{Enc}_{K_1}(\vec{0})$.
  - Otherwise, if $\overline{\mathsf{state}}$ was sent to the adversary earlier as part of a memory request, then the simulator must check the correctness of the response returned by the adversary. To achieve this, the simulator retrieves the $\mathsf{sstorestate}$ at the time $\overline{\mathsf{state}}$ was sent to the adversary. Now, using this $\mathsf{sstorestate}$, the simulator runs the honest sstore instance corresponding to the current subsession to check the correctness of memory request. If the check fails, the simulator simply aborts.
  - If the simulator has not aborted, the simulator retrieves the $\mathsf{oramstate}$ at the time $\overline{\mathsf{state}}$ was sent to the adversary. Now, the simulator calls the ORAM simulator to obtain the next memory request — we assume that the ORAM simulator will return the same answer when invoked with the same $\mathsf{oramstate}$.

    If the next memory request is a write request, then the data for the write is set to a dummy message $\vec{0}$, and then this message is passed along to the sstore instance (which internally performs memory encryption). The outcome of sstore along with a fresh encryption $\overline{\mathsf{state}} := \mathsf{Enc}_{K_1}(\vec{0})$ is sent to the adversary.

Remark 1. *Notice that in the simulation, multiple ciphertexts* $\overline{\mathsf{state}}$ *can correspond to the same point of execution in a subsession. However, if the adversary rewinds the execution of a subsession, all other parts of the response it obtains will be deterministic (i.e., same as the last time) if the simulation does not abort.*

We now show that the view of the adversary in a real execution is indistinguishable from that in the above described simulated execution. We show this indistinguishability between the real and ideal world by using the following sequence of hybrids:

Hybrid H_0: This is the real world execution. The simulator simulates the honest sender and hence, has access to the sender's program. It uses the token $\mathcal{F}_{token}$ to respond to queries by the adversary (receiver).

Hybrid H_1: This hybrid is identical to hybrid H_0 except for the following. For integrity verification, instead of using the Merkle tree scheme, the simulator performs an honest memory check. Merkle tree checks are performed for the memory with Merkle root H', program mem_0 with Merkle root H_S and input with Merkle root H_R.

For each subsession ssid, the simulator maintains a table storing the requests sent by the adversary and the responses sent by the simulator. Specifically, it stores a table consisting of decrypted request state ($\mathsf{oramstate}$ and $\mathsf{sstorestate}$, denoting the execution state), decrypted response state and the snapshot of memory mem'. When $\overline{\mathsf{state}}$ is sent as a part of memory request in an execution query, instead of using Merkle root H', the simulator verifies the correctness by running an honest sstore instance for the subsession. Specifically, the simulator looks up the table by response state and compares the response sent by adversary with mem'. The simulator aborts if the comparison fails. Otherwise, it runs a simulated execution of the token and responds to the adversary.

Recall that as mentioned in the scheme, during an initial linear scan for initializing ORAM, both the program and the input are loaded. When program is loaded during this initialization, the simulator looks up the table based on the decrypted request state and authenticates the program by comparing it to mem_0 instead of using the Merkle root H_S. When the adversary initializes execution, the simulator saves the commitment H_R of the adversary's input in ssid. When the adversary sends a ($\overline{\mathsf{state}}$, input value), the simulator looks up the table by decrypted response state to find the request state sent by the adversary. If it finds an entry, the simulator verifies the correctness of the input value using the Merkle root H_R. *This is where the simulator extracts input from the adversary.* It should be noted that H_R is generated by the adversary. By the collision resistance of hash functions used by Merkle trees, the adversary cannot generate two inputs with the same Merkle root H_R.

Thus, the only event in which this hybrid differs from H_0 is if the adversary breaks the collision resistance of hash functions used by Merkle trees. In this case, the simulator aborts. The probability of this bad event is negligible; this can be shown by reducing the security to a Merkle tree game between a challenger and an adversary. In the absence of this bad event, this hybrid is identical to H_0.

Hybrid H_2: This hybrid is identical to the previous one except for the following. The pseudorandom functions (PRF) are replaced with truly random functions, i.e., whenever PRF function (i) with key K_2 is invoked by the ORAM algorithm and (ii) key K_3 is invoked by sstore in hybrid H_2, the simulator samples a random number instead. In the protocol, the use of a PRF while initializing ORAM ensures that the ORAM accesses memory locations in a deterministic manner. Hence, if the adversary rewinds execution, the same memory locations are accessed by the ORAM algorithm. In this hybrid, we replace PRF with truly random function. Based on the decrypted request state, the simulator determines if the adversary has sent the same query before. If yes, the simulator replays the old answer by using the same randomness in the ORAM algorithm. Otherwise, the simulator looks up the table by response state to determine if this state was sent to the adversary as part of a memory request. If yes, the simulator generates new random numbers to be used by the ORAM algorithm and stores the request and response states along with the random numbers in the table. Except for these changes, the simulator computes the query response as in hybrid H_1. The simulator sends the updated response $\overline{\mathsf{state}}$ to the adversary.

This hybrid is computationally indistinguishable from hybrid H_1 by the security of pseudorandom functions. If the adversary can distinguish between this hybrid and hybrid H_1, we can show a reduction to a game where the adversary can distinguish between a pseudorandom function and a truly random *function* with non-negligible probability.

Hybrid H_3: This hybrid is identical to the previous one except for the following. The simulator replaces the authentication scheme used to verify state with an honest check. In order to do so, along with the other information stored in the table in hybrid H_2, the simulator also stores the encrypted request and response $\overline{\mathsf{state}}$. Note that this is when the simulator starts storing multiple ciphertexts $\overline{\mathsf{state}}$ corresponding to the same point of execution in a subsession.

Instead of using an authentication scheme to verify state, it compares the $\overline{\mathsf{state}}$ sent by the adversary with any of the $\overline{\mathsf{state}}$ values that was previously sent by the simulator. If the simulator does not find an entry in the table, it aborts. If the check succeeds, the simulator can determine the exact execution state based on $\mathsf{oramstate}$ and $\mathsf{sstorestate}$. The simulator knows the program mem_0 from the honest sender, extracts the input inp from the adversary (as described in hybrid H_1) and has stored a snapshot of all random numbers that were generated for ORAM. Hence, by simulating an instance of the token, the simulator can compute the exact response state that needs to be returned without decrypting the request sent by the adversary. The simulator encrypts this response state and sends the encrypted $\overline{\mathsf{state}}$ to the adversary. If the execution proceeds without aborting until time T, then at T-th step, the simulator calls the ideal functionality $\mathcal{F}_{obf}^{\mathcal{RAM}}$ on input inp and sends the output outp to the adversary.

The computational indistinguishability of this hybrid from hybrid H_2 follows from the INT-CTXT se-

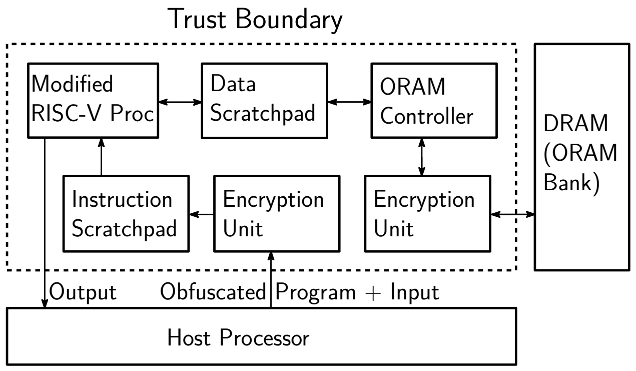

Figure 6: HOP Architecture

curity of the authenticated encryption scheme Enc. If the adversary can distinguish between hybrid H_2 and this hybrid with non-negligible probability, we can show a reduction where the adversary can break the security of an INT-CTXT secure authenticated encryption game with non-negligible probability.

Hybrid H_4**:** This hybrid is identical to the previous one except for the following. The simulator generates a random key K_1. For all $\overline{\mathsf{state}}$ and memory writes by sstore sent to the adversary, the simulator instead sends $Enc_{K_1}(\vec{0})$. Also, the simulator begins execution by sending $\overline{\mathsf{mem}}_0 := \{Enc_{K_1}(\vec{0}\}_{i\in|\mathsf{mem}_0|}$, $\overline{\mathsf{header}} := Enc_{K_1}(\vec{0}||H_s := \mathsf{digest}(\overline{\mathsf{mem}_0}||\mathsf{RAM.params})$ and $\mathsf{RAM.params}$ to the adversary.

Given the security of an IND-CPA secure authenticated encryption scheme Enc, this hybrid is identically distributed as the previous hybrid. If the adversary can distinguish between this hybrid and hybrid H_3, we can show a reduction where the adversary can break the security of an IND-CPA secure authenticated encryption scheme.

Hybrid H_5**:** This hybrid is identical to the previous one except for the following. Instead of the execution of an ORAM algorithm, the simulator invokes an ORAM simulator. For an execution query, after checking the correctness of the response sent by the adversary, the simulator retrieves the $\mathsf{oramstate}$ and invokes the ORAM simulator with this state. We assume that the ORAM simulator will return the same answer when invoked with the same $\mathsf{oramstate}$. The output of the ORAM simulator is sent to the adversary. Assuming that the ORAM scheme is statistically secure, this hybrid is statistically indistinguishable from hybrid H_4.

In this hybrid, the simulator uses an ORAM simulator for ORAM requests, performs ideal checks for the memory that needs to be stored by the adversary and for the token state sent to the adversary, uses truly random functions, and sends $Enc_{K_1}(\vec{0})$ to the adversary. The simulator knows the program mem_0 from the sender, extracts the program input inp from the adversary as described in hybrid H_1. If the execution proceeds until T steps without aborting, the simulator internally simulates an instance of the ideal functionality $\mathcal{F}_{obf}^{\mathcal{RAM}}$ to obtain the output outp. The simulator in this hybrid behaves exactly as the ideal world simulator described earlier. Hence, this is the ideal world execution.

5 Implementation

The final architecture of HOP (with the optimizations from Section 3) is shown in Figure 6. We now describe implementation-specific details for each major component.

5.1 Modified RISC-V Processor and Scratchpad

We built HOP with a RISC-V processor which implements a single stage 32bit integer base user-level ISA developed at UC Berkeley [12]. A RISC-V C cross-compiler is used to compile C programs to be run on the processor. The RISC-V processor is modified to include a 16 KB instruction scratchpad and a 512 KB data

```
 1: int decompress(char *chunk) {
 2:      int compLen = 0;
 3:     // initial processing
 4:     burrowsWheeler(chunk, compLen);
 5:     // more processing
 6:     writeOutput(chunk);
 7:      return compLen;}
 8: void main() {
 9:     char *inp = readInput();
10:     for (i = 0; i < len(inp); i += len) {
11:        spld(inp + i, CSIZE, 0);
12:         len = decompress(inp + i); } }
```

Figure 7: Example program using **spld**: `bzip2`

scratchpad (Section 3.4). The RISC-V processor and the compiler are modified accordingly to accommodate the new scratchpad load/unload instructions (described below). While HOP uses a single stage RISC-V processor, our system does not preclude additional hardware optimizations in commodity processors such as multi-issue, branch predictor, etc. Our only requirement to support such processor structures is the ability to calculate, for that program over all inputs, a suitably conservative maximum runtime T.

New scratchpad instructions. For our prototype, we load the scratchpad using a new instruction called **spld**, which is specified as follows:

$$\textbf{spld}\ addr, \#mem, spaddr$$

In particular, *addr* is used to specify the starting address of the memory that needs to be loaded in scratchpad. *#mem* is the number of memory locations to be loaded on the scratchpad starting at *addr* and *spaddr* is the location in scratchpad to store the loaded data. When the processor intercepts an **spld** instruction, it performs two operations: 1. It writes back the data stored in this scratchpad location to the appropriate address in main memory (ORAM). 2. It reads *#mem* memory locations starting at main memory address *addr* into scratchpad locations starting at *spaddr*. Of course, **spld**'s precise design is not fundamental: we need a way to load an on-chip memory such that it is still feasible to statically determine T.

Example scratchpad use. Figure 7 shows an example scenario where **spld** is used. The program shows a part of the code used for decompressing data using the `bzip2` compression algorithm. The algorithm decompresses blocks of compressed data and outputs data of size CSIZE independently. Each block of data may be read and processed multiple times during different steps of compression (run-length encoding, Burrows-Wheeler transform, etc.). Hence, each such block is loaded into the scratchpad (line 11) before processing. This ensures that every subsequent access to this data is served by the scratchpad instead of memory (thereby reducing expensive ORAM accesses). After decompressing the block, **spld** is executed for the next block of compressed data.

5.2 ORAM Controller

We use a hardware ORAM controller called 'Tiny ORAM' from [19, 20]. The ORAM controller implements an ORAM tree with 25 levels, having 4 blocks per bucket. Each block is 512 bits (64 Bytes) to match modern processor cache line size. This corresponds to a total memory of 4 GB. The ORAM controller uses a stash of size 128 blocks and an on-chip position map of 256 KB. For integrity and freshness, Tiny ORAM uses the PosMap MAC (PMMAC) scheme [19]. We note that PMMAC protects data integrity but does not achieve malicious security. We estimate the cost of malicious security using a hardware Merkle-tree on ORAM in Table 2. We disable the PosMap Lookaside Buffer (PLB) in Freecursive ORAM to avoid leakage through the total number of ORAM accesses.

Table 2: **Resource allocation and utilization of HOP on Xilinx Virtex V7485t FPGA.** For each row, first line indicates the estimate. % utilization is mentioned in parentheses. LUT: Slice LookUp Table, FFs: Flip-flops or slice registers, BRAM: Block RAM.

	LUT	FFs	LUT-Mem	BRAM
Total Estimate	169472	51870	81112	566.5
(% Utilization)	(55.8%)	(8.5%)	(62.0%)	(55.0%)
HOP Estimate	103462	39803	38725	437
(% Utilization)	(34.0%)	(6.6%)	(47.7%)	(42.4%)
(HOP− ORAM) Estimate	21626	6579	1	83
(% Utilization)	(7.1%)	(1.1%)	(∼0%)	(8.1%)
Estimate with Merkle tree	221041	81410	81126	566.5
(% Utilization)	(72.8%)	(13.4%)	(62.0%)	(55.0%)

5.3 Encryption Units

For all encryption units, we use tinyaes from OpenCores [2]. The encryption units communicate with the external DRAM (bandwidth of 64 Bytes/cycle) as well as the host processor. Data is encrypted before writing to the DRAM. Similarly, all data read from the DRAM is decrypted first before processed by the ORAM controller. Another encryption unit is used to decrypt the obfuscated program before loading it into the instruction scratchpad.

6 Evaluation

We now present a detailed evaluation of HOP for some commonly used programs, and compare HOP to prior work.

6.1 Methodology

We measure program execution time in processor cycles, and compare with our own baseline scheme (to show the effectiveness of our optimizations), an insecure processor as well as related prior work. For each program, we choose parameters so that our baseline scheme requires about 100 million cycles to execute. We also report processor idle time, the time spent on dummy arithmetic instructions and dummy memory accesses to adhere to an $A^N M$ schedule (Section 3.3).

For the programs we evaluate (except *bzip2*; c.f., Section 6.4), we calculate T manually. We remark that the average input completion time and worst case time are very similar for these programs. To find T for larger programs, one may use established techniques in determining worst case execution time (e.g., a tool from [54]).

In our prototype, evaluating an arithmetic instruction takes 1 cycle while reading/writing a word from the scratchpad takes 3 cycles. Given the parameters in Section 5.2, an ORAM access takes 3000 cycles. For our HOP configurations with a scratchpad, we require both scratchpad read/writes and arithmetic instructions to take 3 cycles in order to hide which is occurring. Following Section 3.3, we set $N = 3000$ when not using a scratchpad; with a scratchpad, we use $N = 1000$. For our evaluation, we consider programs ranging from those with high locality (e.g., `bwt-rle`) to those that show no locality (e.g., `binsearch`).

6.2 Area Results

We synthesized, placed and routed HOP on a Xilinx Virtex V7485t FPGA for parameters described in Section 5. HOP operates at 79.3 MHz on this FPGA. The resource allocation and utilization figures are mentioned in Table 2. The first three rows represent the total estimate, estimate for HOP (i.e. excluding RISC-Vprocessor, and the scratchpad) and an estimate for HOP that does not account for ORAM. The last row shows the total overhead including an estimate for a Merkle tree scheme. Excluding the processor, scratchpad and ORAM, HOP consumes $< 9\%$ of the FPGA resources. We see that the total area overhead of HOP is small and can be built on a single FPGA chip.

6.3 Main Results

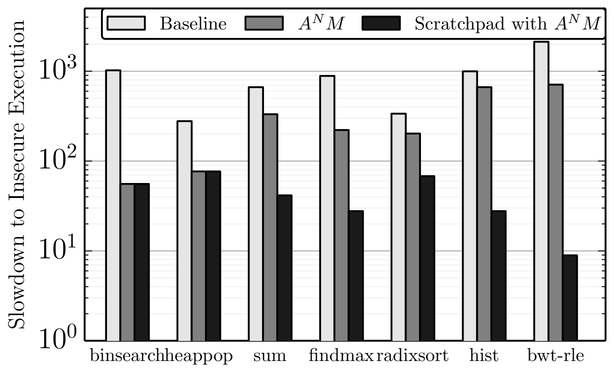

Figure 8: Execution time for different programs with (i) baseline scheme, (ii) $A^N M$ schedule and (iii) Scratchpad + $A^N M$.

Figure 8 shows the execution time of HOP variants relative to an insecure processor. For each program, there are three bars shown. The first bar is for the baseline HOP scheme (i.e., Section 3.2 only); the second bar only uses an $A^N M$ schedule without a scratchpad (adds Section 3.3); and the third bar is our final scheme that uses a scratchpad and the $A^N M$ schedule (adds Section 3.4). All schemes are relative to an insecure processor that does not use ORAM or hide what instruction it is executing. We assume this processor uses a scratchpad that has the same capacity as HOP in Section 5.1. The time required to insert the program and data is not shown.

Comparison of HOP variants. As can been seen in the figure, the $A^N M$ schedule without a scratchpad gives a $1.5\times \sim 18\times$ improvement. Adhering to an $A^N M$ schedule requires some dummy arithmetic or memory instructions during which the processor is essentially idle. We observe that for our programs, the idle time ranges between 43% and 49.9% of the execution time, consistent with the claim in Section 3.3.

Effect of a scratchpad. The effect of a scratchpad largely depends on program locality. We thus classify programs in our evaluation into four classes:

1. Programs such as `binsearch, heappop` do not show locality. Thus, a scratchpad does not improve performance.
2. Programs such as `sum, findmax` stream (linear scan) over the input data. Given that an ORAM block is larger than a word size (512 bits vs 32 bits in our case), a scratchpad in these streaming applications can serve the next few (7 with our parameters) memory accesses after **spld**. A larger ORAM block size can slightly benefit these applications while severely penalize programs with no locality, and therefore is not a good trade-off.
3. Programs that maintain a small working set at all times will greatly benefit from a scratchpad. We evaluate one such program `bwt-rle`, which performs Burrows-Wheeler transform and run length encoding, and is used in compression algorithms.
4. Lastly, some programs are a mix of the above cases — some data structures can be entirely loaded into the scratchpad whereas some cannot (e.g. a Radix sort program).

Comparison to insecure processor. The remaining performance overhead of the optimized HOP (the third bar) comes from several sources. First, the performance of ORAM: The number of cycles to perform a memory access using ORAM is much higher than a regular DRAM. In HOP, an ORAM access is 40× more expensive than an insecure access. Second, dummy accesses to adhere to a schedule: As shown in Section 3.3, the performance overhead due to dummy accesses $\leq 2\times$. For programs such as `bwt-rle`, HOP has a slowdown as low as 8×. This is primarily due to the reduction in ORAM accesses by maintaining a small working set in the scratchpad.

6.4 Case Study: bzip2

To show readers how our system performs on a realistic and complex benchmark, we evaluate HOP on the open-source algorithm `bzip2` (re-written for a scratchpad, cf. Figure 7). We evaluate the decompression algorithm only, as the decompression algorithm's performance does not heavily depend on the input if one fixes the input size [1]. This allows us to run an average case input and use its performance to approximate the effect of running other inputs. To give a better sense for how the optimizations are impacted by different inputs, we don't terminate at a worst-case time T but rather terminate as soon as the program completes.

We run tests on two inputs, both highly compressible strings. For the first input, HOP achieves 106× speedup over the baseline scheme and 17× slowdown over the insecure version. For the second input, HOP achieves 234× speedup over the baseline and 8× slowdown over the insecure version. Thus, the gains and slowdowns we see from the prior studies extend to this more sophisticated benchmark.

6.5 Comparison with Related Work

We now compare against prior work on obfuscation with hardware (these prior works were not implemented) and several works with related threat models.

6.5.1 Comparison to prior obfuscation from trusted hardware proposals [15, 17, 27]

We now compare against [15, 17, 27] which describe obfuscation using trusted hardware. Note that none of these schemes were implemented.

Part of the proposals in [15, 17] require programs to be run as *universal circuits* under FHE while [27] evaluates programs as universal circuits directly on hardware (i.e., by feeding the encrypted inputs of each gate into a stateless hardware unit: where it decrypts the inputs, evaluates the gate, and re-encrypts the output). We will now compare HOP to these circuit-based approaches. Again, we stress that all of [15, 17, 27] require the use of trusted hardware for their complete scheme and thus can be viewed similarly to HOP from a security perspective.

Table 3 shows the speedup achieved by HOP relative to universal circuits run under FHE (left) and bare hardware (right). We assume the cost of a universal circuit capable of evaluating any c gate circuit is $18 * c * \log c$ gates [39]. We compare the approaches on the `findmax` and `binsearch` benchmarks, using a dataset size of 1 GB for each. We show `findmax` as it yields a very efficient circuit and a best-case situation for the circuit approach (relative to the corresponding RAM program); `binsearch` shows the other extreme. For [15, 17], we assume a BGV-style FHE scheme [9], using the NTRU cryptosystem, with polynomial dimension and ciphertext space parameters chosen using [24], to achieve 80 bits of security.[6] For [27], we assume each NAND gate takes 20 cycles to evaluate (10 cycles for input decryption with AES, 0 cycles for evaluation, 10 cycles for re-encryption). For HOP, we assume the parameters from Section 5.

In the Table, On+Off ('online and offline') assumes one search query is run: in that case, HOP's performance is reduced due to the time needed to initially load the ORAM. The On ('online only') column shows the amortized speedup when many search queries are made without changing the underlying search database

[6]When represented as circuits, both `findmax` and `binsearch` look like a linear PIR. Over a 1 GByte dataset, we evaluate this function with a 10-level FHE circuit, which gives an FHE polynomial dimension (n) of ~ 8192 and ciphertext space q of $\sim 2^{128}$ (using terminology from [9]). With these parameters, a single polynomial multiplication/addition using NTL [50] costs 14 ms / .4 ms on a 3 GHz machine.

	FHE [15, 17]		Hardware [27]	
	On + Off	On	On + Off	On
`findmax`	$1 * 10^9$	$2 * 10^9$	$4 * 10^3$	$1 * 10^4$
`binsearch`	$4 * 10^9$	$4 * 10^{15}$	$6 * 10^3$	$1 * 10^{10}$

Table 3: HOP speedup (×) relative to universal circuit approaches. `findmax` and `binsearch` are over 1 GB datasets.

(i.e., without re-loading the ORAM each time). This shows an inherent difference to works based on universal circuits: those works represent programs as circuits, where optimized algorithms such as `binsearch` do not see speedup. In all cases, HOP shows orders of magnitude improvement to the prior schemes.

We note that our comparison to [15, 17] is conservative: we only include FHE's time to perform AND/OR gate operations and not the cost of auxiliary FHE operations (re-linearization, modulus switching, bootstrapping, etc). Lastly, FHE is only one part of [15, 17]: we don't include the cost of NIZK protocols, etc. which those schemes also require.

6.5.2 Comparison with $i\mathcal{O}$ [37]

We compare HOP with an implementation of indistinguishability obfuscation ($i\mathcal{O}$) that does not assume a trusted hardware token. Note that while VBB obfuscation is not achievable in general, $i\mathcal{O}$ is a weaker notion of obfuscation. With [37], evaluating an 80-bit point function (a simple function that is 0 everywhere except at one point) takes about 180 seconds while HOP takes less than a msec, which is about 5-6 orders of magnitude faster.

6.5.3 Comparison with GhostRider [40]

Recall from Section 2 that GhostRider protects input data to the program *but not the program.* Since our privacy guarantee is strictly greater than GhostRider, we now compare to that work to show the cost of extra security. Note: we compare to the GhostRider compiler and not the implementation in [40] which uses a different parameterization for the ORAM scheme. This comparison shows the additional cost that is incurred by HOP to hide the program. We don't show the full comparison for lack of space, but point out the following extreme points: For programs with unpredictable access patterns (`binsearch, heappop`), GhostRider outperforms HOP by $\sim 2\times$. HOP's additional overhead is from executing dummy instructions to adhere to a particular schedule. For programs with predictable access patterns (`sum, findmax, hist`), GhostRider's performance is similar to that of an insecure processor.

6.6 Time for Context Switch

Since it was not required for our performance evaluation, we have not yet implemented context switching (Section 3.5) in our prototype. Recall, context switching means the receiver interrupts the processor, which encrypts and writes out all the processor state (including CPU state, instruction scratchpad, data scratchpad, ORAM position map and stash) to DRAM. We estimate the time of a context switch as follows. The total amount of data stored by our token is $\sim$ 800 KB (Section 5). Assuming a DRAM bandwidth of 10 GB/s and a matching encryption bandwidth, it would take $\sim 160\mu$s to perform a context switch to run another program. Note that this assumes all data for a swapped-out context is stored in DRAM (i.e., the ORAM data already in the DRAM need not be moved). If it must be swapped out to disk because the DRAM must make room for the new context, the context switch time grows proportional to the ORAM size.

7 Conclusion

This paper makes two main contributions. First, we construct an optimized hardware architecture - called HOP - for running obfuscated RAM programs. We give a matching theoretic model for our optimized architecture and prove it secure. A by-product of our analysis shows the first obfuscation for RAM programs using 'stateless' tokens. Second, we present a complete implementation of our optimized architecture and evaluate it on real-world programs. The complete design requires 72% the area of a V7485t Field Programmable Gate Array (FPGA) chip. Run on a variety of benchmarks, HOP achieves an average overhead of $8\times \sim 76\times$ relative to an insecure system. To the best of our knowledge, this effort represents the *first* implementation of a provably secure VBB obfuscation scheme in any model under any assumptions.

References

[1] bzip2 man pages. `http://www.bzip.org/1.0.5/bzip2.txt`.

[2] Open cores. `http://opencores.org/`.

[3] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The em sidechannel (s). In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 29–45. Springer, 2002.

[4] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *CRYPTO*, 2001.

[5] N. Bitansky, R. Canetti, S. Goldwasser, S. Halevi, Y. T. Kalai, and G. N. Rothblum. Program obfuscation with leaky hardware. In *Advances in Cryptology–ASIACRYPT 2011*, pages 722–739. Springer, 2011.

[6] N. Bitansky, S. Garg, H. Lin, R. Pass, and S. Telang. Succinct randomized encodings and their applications. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, pages 439–448. ACM, 2015.

[7] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *IEEE S&P*, 2014.

[8] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In *International Conference on the Theory and Applications of Cryptographic Techniques*, 1997.

[9] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *ITCS*, 2012.

[10] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[11] R. Canetti, J. Holmgren, A. Jain, and V. Vaikuntanathan. Succinct garbling and indistinguishability obfuscation for RAM programs. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, pages 429–437. ACM, 2015.

[12] C. Celio and E. Love. The sodor processor collection. `http://riscv.org/download.html#tab_sodor`.

[13] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *HPCA*, 2010.

[14] J. H. Cheon, K. Han, C. Lee, H. Ryu, and D. Stehlé. Cryptanalysis of the multilinear map over the integers. In *EUROCRYPT*. 2015.

[15] K.-M. Chung, J. Katz, and H.-S. Zhou. Functional encryption from (small) hardware tokens. In *ASIACRYPT*. 2013.

[16] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[17] N. Döttling, T. Mie, J. Müller-Quade, and T. Nilges. Basing obfuscation on simple tamper-proof hardware assumptions. *IACR Cryptology ePrint Archive*, 2011.

[18] C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.

[19] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas. Freecursive oram:[nearly] free recursion and integrity verification for position-based oblivious ram. In *ASPLOS*, 2015.

[20] C. W. Fletcher, L. Ren, A. Kwon, M. Van Dijk, E. Stefanov, D. Serpanos, and S. Devadas. A low-latency, low-area hardware oblivious ram controller. In *FCCM, 2015*. IEEE, 2015.

[21] S. Garg. Program obfuscation via multilinear maps. In *Security and Cryptography for Networks*. 2014.

[22] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *FOCS*, 2013.

[23] C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs. Garbled ram revisited. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2014.

[24] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. In *Advances in Cryptology–CRYPTO 2012*. 2012.

[25] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *ACM symposium on Theory of computing (STOC)*, 1987.

[26] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 1996.

[27] V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia. Founding cryptography on tamper-proof hardware tokens. In *TCC*. 2010.

[28] D. Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 1st edition, 2009.

[29] S. Hada. Zero-knowledge and code obfuscation. In *ASIACRYPT 2000*, 2000.

[30] K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs. In *CHES 2010*. 2010.

[31] M. Kainth, L. Krishnan, C. Narayana, S. G. Virupaksha, and R. Tessier. Hardware-assisted code obfuscation for fpga soft microprocessors. In *Design, Automation & Test in Europe Conference & Exhibition*, 2015.

[32] J. Katz. Universally composable multi-party computation using tamper-proof hardware. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2007.

[33] Á. Kiss and T. Schneider. Valiant's universal circuit is practical. In *EUROCRYPT*. 2016.

[34] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO'99*, 1999.

[35] V. Kolesnikov and T. Schneider. A practical universal circuit construction and secure evaluation of private functions. In *FCDS*. 2008.

[36] V. Koppula, A. B. Lewko, and B. Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, pages 419–428. ACM, 2015.

[37] K. Lewi, A. J. Malozemoff, D. Apon, B. Carmer, A. Foltzer, D. Wagner, D. W. Archer, D. Boneh, J. Katz, and M. Raykova. 5gen: A framework for prototyping applications using multilinear maps and matrix branching programs. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 981–992. ACM, 2016.

[38] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 2000.

[39] H. Lipmaa, P. Mohassel, and S. Sadeghian. Valiant's universal circuit: Improvements, implementation, and applications. Cryptology ePrint Archive, Report 2016/017, 2016. `http://eprint.iacr.org/2016/017`.

[40] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*, 2015.

[41] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. In *CCS*, 2013.

[42] 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@ ISCA*, 2013.

[43] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. Observing and preventing leakage in mapreduce. In *CCS*, 2015.

[44] R. Pass, E. Shi, and F. Tramr. Formal abstractions for attested execution secure processors.

[45] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, 2015.

[46] L. Ren, X. Yu, C. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *Symposium on Computer Architecture*, 2013.

[47] S. Schrittwieser and S. Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *Information Hiding*, 2011.

[48] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *IEEE S&P*, 2015.

[49] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with $O(\log^3 N)$ worst-case cost. In *ASIACRYPT*, 2011.

[50] V. Shoup et al. Ntl: A library for doing number theory, 2001.

[51] E. M. Songhori, S. Zeitouni, G. Dessouky, T. Schneider, A.-R. Sadeghi, and F. Koushanfar. Garbledcpu: A mips processor for secure computation in hardware. In *DAC*. 2016.

[52] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM – an extremely simple oblivious ram protocol. In *CCS*, 2013.

[53] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Conference on Supercomputing*, 2003.

[54] L. Tan. The worst case execution time tool challenge 2006: The external test. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006.*, 2006.

[55] L. G. Valiant. Universal circuits (preliminary report). In *STOC*, 1976.

[56] X. S. Wang, S. D. Gordon, A. McIntosh, and J. Katz. Secure computation of mips machine code. Technical report, ePrint 2015/547, 2015.

[57] X. Zhuang, T. Zhang, and S. Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. In *ACM SIGPLAN Notices*, 2004.

A Obfuscation in the Public-Key Setting

For the sake of simplicity, we describe our construction and proof in the model where a single sender embeds a symmetric key into a secure processor and provides this to the receiver along with the obfuscated program to execute. However, we note that we can extend our results to reuse the token and allow multiple senders to obfuscated the program for a receiver. For example, suppose two senders S_1 and S_2 would like to both send encrypted programs to be executed by a receiver R on a hardware token (provided by a trusted hardware manufacturer). The hardware would then be initialized with a secret key $\mathsf{sk}_{\mathsf{enc}}$ of a public-key CCA secure encryption scheme (with public key $\mathsf{pk}_{\mathsf{enc}}$) along with a verification key $\mathsf{vk}_{\mathsf{sig}}$ of a signature scheme (with signing key $\mathsf{sk}_{\mathsf{sig}}$). The signing key $\mathsf{sk}_{\mathsf{sig}}$ would be owned by a trusted certificate authority and would also be stored in the token. Now, in our construction, we would replace the symmetric key CCA secure authenticated encryption with a public key CCA secure encryption, where all ciphertext are authenticated with a signature scheme. When S_1 wishes to send an obfuscated program P_1 to a receiver R, S_1 would pick a signing key/verification key pair $(\mathsf{sk}_{S_1}, \mathsf{vk}_{S_1})$. S_1 will obtain a signature of vk_{S_1} from the trusted certificate authority (denote this signature by σ and note that this signature will verify under the verification key $\mathsf{vk}_{\mathsf{sig}}$). Now, S_1 will encrypt P_1 with $\mathsf{pk}_{\mathsf{enc}}$ and authenticate all ciphertexts with sk_{S_1} and provide these ciphertexts along with σ to the receiver. The receiver will feed in encrypted ciphertexts along with σ to the token. The token, when decrypting ciphertexts, will first check the validity of vk_{S_1} by verifying σ and the signatures of all the ciphertexts. If all the checks pass, the token will decrypt the ciphertexts using $\mathsf{sk}_{\mathsf{enc}}$. When encrypting state to be sent back to the receiver, the token will encrypt it with $\mathsf{pk}_{\mathsf{enc}}$ and sign it with $\mathsf{sk}_{\mathsf{sig}}$. This will mimic the symmetric key CCA secure authenticated encryption scheme that we use in our single sender/receiver scheme.