HOP: Hardware makes Obfuscation Practical

Kartik Nayak*, Christopher W. Fletcher†, Ling Ren‡, Nishanth Chandran§, Satya Lokam§, Elaine Shi∥ and Vipul Goyal§

*UMD – kartik@cs.umd.edu
†UIUC – cwfletch@illinois.edu
‡MIT – renling@mit.edu
§Microsoft Research – {nichandr, satya, vipul}@microsoft.com
∥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 theoretical 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 8x × 76x 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.

I. 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 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

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author’s employer if the paper was prepared within the scope of employment.

NDSS ’17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23349
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. 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 (clear-text) 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 and execute operations can be decoupled 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 B. 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 III-B) 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.

A. 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.\footnote{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 III.} 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 \textit{rewinding} and \textit{mix-and-match} attacks (which 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 \textit{program privacy} and exhibit no program-dependent behavior. With these optimizations, HOP performs $5 \times 238 \times$ better than the baseline HOP design across simple to sophisticated programs while the overhead over an insecure system is $8 \times 76 \times$. 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 III-C).

II. RELATED WORK

Obfuscation. The formal study of virtual black-box (VBB) obfuscation was initiated by Hada [29] and Barak \textit{et al.} [4]. Unfortunately, Barak \textit{et al.} showed that it is impossible to achieve program obfuscation for general programs. Barak \textit{et al.} also defined a weaker notion of indistinguishability obfuscation ($iO$), which avoids their impossibility results. Garg \textit{et al.} [22] proposed a construction of $iO$ for all circuits based on assumptions related to multilinear maps. However, these constructions are not efficient from a practical standpoint. There are constructions for $iO$ 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 \textit{et al.} [27] considered virtual black-box obfuscators on minimal secure hardware tokens. Goyal \textit{et al.} show how to achieve VBB obfuscation for all polynomial time computable functions using \textit{stateless} secure hardware tokens that only perform authenticated encryption/decryption and a single NAND operation. In a related line of work, Döttling \textit{et al.} [17] show a construction for program obfuscation using a single stateless hardware token in universally input-oblivious models of computation. Bitansky \textit{et al.} [5] show a construction for program obfuscation from “leaky” hardware. Similarly, Chung \textit{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 VI-E.

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], [58].

There have been significant 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 \textit{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 VI-E.

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 [58], [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 III-E), 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.

### III. 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 IV.

#### A. 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.

#### B. 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 IV-A.)

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 III-A.

**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. 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.

C. 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 \( A^4 M \), 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: 

\[ \text{A A A A M A A M} \]

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 III-B. 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 Appendix A for a proof of this claim. We consider this proof to be of independent interest. 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.

D. 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. It is used to temporarily store a portion of the 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 V-A. 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., they 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 to 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 III-C.

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.

---

3Our ORAM latency from Section VI 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.

4We 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.
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 VI-C).

### E. 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 \( \text{state} \)) 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, to implement, in reality it introduces avenues for new attacks.

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 \( \text{state} \)) 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.

### IV. Formal Scheme

We now give a formal model for the fully optimized HOP processor (i.e., including all subsections in Section III) and prove its security in UC framework. Section IV-A describes the preliminaries. Section IV-B describes the ideal functionality for obfuscation of RAM programs. Sections IV-C and IV-D describe our formal scheme and proof in the UC framework.

#### A. Preliminaries

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

Program output

Digest of sender's program, i.e., bit-lengths of input, output, and memory word

State stored by

Number of words in memory

Time for program execution

Merkle root of the main memory

As a result, \( \text{cpustate} \) denotes the CPU's initial internal state, and \( \text{RAM} \) the bit-width of a memory word; and \( \text{cpustate}' \) denotes a family of \( \text{RAM} \) programs with the following public parameters:

\( \Pi, T, N, \ell_{in}, \ell_{out}, w \)

\( \text{RAM} \) supports two types of operations: a) On \( \text{read} \), it outputs \( \text{mem}[\text{addr}] \); b) On \( \text{write} \), it sets \( \text{mem}[\text{addr}] := \text{wdata} \), and outputs \( \bot \). In this paper, we define an Oblivious \( \text{RAM} \) as a \textit{stateful}, probabilistic algorithm that interacts with a memory array \( \text{mem} \). It is denoted as \( \text{ORAM}^{N,w} \) where \( N \) and \( w \) are public parameters denoting the memory capacity in terms of number of words, and the bit-width of a word. \( \text{mem} \) denotes the initial state of the memory, where all but the first \( N \) locations are set to 0. An \( \text{ORAM} \) converts memory contents \( \text{mem} \) to \( \text{mem}' \). An \( \text{ORAM} \) takes two types of inputs: \( \text{op} := (\text{read}, \text{addr}) \), and \( \text{op} := (\text{write}, \text{addr}, \text{wdata}) \). After receiving input \( \text{op}_i \), \( \text{ORAM} \) interacts with \( \text{mem}' \), and produces read/write operations into \( \text{mem}' \) as output, denoted by \( \text{op}_i \). These operations \( \text{op}_i \) implicitly define memory contents of \( \text{mem} \).

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

\textbf{Remark: ORAM initialization.} In this paper, we assume an \( \text{ORAM} \) initializes with a memory array where the first \( N \) words are non-zero (reflecting the initial unshuffled memory), followed by all zeros. Most \( \text{ORAM} \) schemes require an initialization procedure to shuffle the initial memory contents. In this paper, we assume that the \( \text{ORAM} \) algorithm performs a linear scan of first \( N \) memory locations and inserts them into \( \text{ORAM} \). 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 \( \text{ORAM} \) algorithm upon the first read or write operation — therefore our notation does not make such initialization explicit. This also means that the first \( \text{ORAM} \) operation will incur a higher overhead than others.
The sender creates an encrypted header \( \text{header} \) by encrypting the session identifier \( \text{ssid} \) with key \( K \).

\[
\text{header} := \text{Enc}_{K}(\text{ssid})
\]

Then the receiver decrypts the header using the same key:

\[
\text{ssid} := \text{Dec}_{K}(\text{header})
\]

Now the receiver uses the session identifier \( \text{ssid} \) to create a new instance of the functionality \( F_{\text{internal}} \). The functionality stores the new session identifier \( \text{ssid} \) and the program, and sets its state to the decrypted state.

\[
\text{state} := \text{Enc}_{K}(\text{state})
\]

The receiver then sends the state to the sender.

\[
\text{state} \rightarrow \text{sender}
\]

The sender receives the state and uses it to execute the program. The execution process involves multiple steps, including reading, writing, and aborting.

\[
\text{execute} \rightarrow \text{sender}
\]

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 implementing functionality \( F_{\text{token}} \) to the receiver. The receiver can use the same stateless token to execute multiple obfuscated programs sent by the sender.
Sender: On receive (“create”, RAM = (cpustate0, mem0)) from env:
1: If not initialized: K := (K1, K2, K3) 8 (0, 1)3λ, send
   (“store key”, K) to Ftoken, await “done”
2: mem0 := ⟨EncK1(mem0[i])⟩i∈[mem0]  
3: HS := digest(mem0) // HS: program Merkle root
4: header := EncK1(cpustate0||HS||RAM.params)
5: Send (header, mem0, RAM.params) to receiver

Receiver: On receive (“execute”, pid, inp) from env:
1: Await (header, mem0, RAM.params) from sender s.t.
   RAM.params, HS = pid if not received already
2: Initialize mem := mem0 || inp || 0
3: Send (“initialize”, header, H′ := digest(inp)) to Ftoken,
   await state from Ftoken
4: for i in {1, . . . , T}:
5: Send (“execute one step”, state) to Ftoken
6: Await (oper, state) from Ftoken; state overwritten with
   the received value
7: Until oper = (“okay”, _), repeat: //multiple memory
   requests for the RAM step due to ORAM
8: perform the operation oper on mem and let the
   response be res
9: forward (res, state) to Ftoken, and await
   (oper, state) from Ftoken
10: Parse oper := (“okay”, outp), output outp

Fig. 4: Protocol Protobj. Realizes FRAM in the Ftoken-
hybrid model.

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 [κ := PRFK2(ssid), oramstate]
   (where ssid := (HS, H′)) 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 men-
   tioned in Section IV-A, we assume that the ORAM initial-
   ization 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 de-
   terministic secure storage module that sits 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 κ := PRFK3(ssid) and sstorestate. 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, sstorestate is initialized
to sstorestate := (HS, H′, H′ := 0), where HS denotes
   the Merkle root of the encrypted program provided by the
   sender, H′ 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 (read, addr) or (write, addr, wdata):
   • If addr is in the mem0 part of the memory (the sender-
     provided encrypted program), interact with mem and use
     HS to verify responses. Update HS appropriately if the
     request type is write.
   • If addr is in the inp part of the memory (the receiver-
     provided input), interact with mem and use H′ 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 III-C and III-D). RRAM can be
   augmented to use a next instruction circuit Π′ := ΠN
   for a fixed N, with the following modifications:
a) Π′ is a combinational circuit, which consists of N next-
   instruction circuits Πi cascaded as shown in Figure 5.
b) The Πi’s use an additional shared memory, referred to as
   a scratchpad. Each Πi (except Π1) operates on the output
   of Πi−1 and an operand rdta−1 read from scratchpad.
   The next instruction circuit Π′ outputs opN to retrieve
   rdta from mem, which is subsequently used by Π1.
   On input inp, the execution of RAM[T, N, ℓin, ℓout, w] :=
   (Π′, cpustate, mem) is similar to what was defined in
   Section IV-A but uses Π′ as the next instruction circuit. The
   augmented random access machine RRAM′ models a RAM
   that performs N instructions followed by an ORAM access.
   If some opi cannot be served by the scratchpad, subsequent
   Πj for i + 1 ≤ j ≤ N do not update cpustatei and output
   opN = opi to load the required data in scratchpad.

Remark. For augmented random access machines that uses
a scratchpad, rdta 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
As mentioned in Section IV-A, during initialization, \( F_{\text{internal}} \) executes the next instruction circuit \( I' \) of the RAM machine to obtain an updated \( \text{cpustate} \) and an \( \text{op} \in \{\text{read, write}\} \). Once operation \( \text{op} \) is performed by the ORAM algorithm, \( F_{\text{internal}} \) updates \( \text{state}.time \) to reflect the execution of the instruction (Figure 3 line 5). The message “okay” is then sent to the receiver. At time \( t = T \), \( F_{\text{internal}} \) returns the program output to the receiver (Figure 3 line 6).

- **Execute one step**: This is shown in Figure 3 and Figure 4 line 5. When this query is invoked, \( F_{\text{internal}} \) executes the next instruction circuit \( I' \) and forwards the request to \( F_{\text{token}} \) (Figure 4 lines 7-9). To account for instantiation of any ORAM, \( F_{\text{token}} \) is shown to receive any query from receiver (indicated by wildcard \( \_ \) in Figures 3 and 4). These queries are sent to \( F_{\text{internal}} \) 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 \( \text{inp} \) in its memory \( \text{mem} := \text{mem}_0||\text{inp}||0 \). It commits to its input by invoking “initialize” (Figure 4 line 3) and sending a Merkle root of its input \( (H_R = \text{digest(}\text{inp}\text{)}) \) along with header \( := \text{Enc}_{K_1}(\text{cpustate}_0||H_S||\text{RAM.params}) \). \( F_{\text{token}} \) initializes the parameters, creates \( \text{state} \) 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 IV-A, during initialization, ORAM in \( F_{\text{token}} \) reads \( \text{mem}_0 \) word by word (not shown in figure). For each word read, sstore performs Merkle tree verification with \( H_S := \text{digest(}\text{mem}\text{)} \). Similarly, when the input is read, sstore verifies it with \( H_R := \text{digest(}\text{inp}\text{)} \). In particular, \( \text{state} \) and \( \text{mem}, \text{spaddr} \) are unique and are encrypted by the processor to prevent any interference from the attacker.

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

\[ \text{spld} \ \text{addr}, \#\text{mem}, \text{spaddr} \]

In particular, \( \text{addr} \) is used to specify the starting address of the memory that needs to be loaded in scratchpad. \( \#\text{mem} \) is the number of memory locations to be loaded on the scratchpad. When the processor intercepts an \( \text{spld} \) instruction, it performs two operations: 1. It writes the data stored in this scratchpad location to the appropriate address in the memory. 2. It reads \( \#\text{mem} \) memory locations starting at main memory address \( \text{addr} \) into scratchpad locations starting at \( \text{spaddr} \). Of course, \( \text{spld} \)'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 \( T \).

**D. Proof of Security**

**Theorem 1.** Assuming that \( \text{Enc} \) is an INT-CTXT + IND-CPA authenticated encryption scheme, ORAM satisfies obliviousness (Section IV-A), 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 \( F_{\text{obj}^{\text{RAM}}} \) (Figure 2) in the \( F_{\text{token}} \)-hybrid model.

**Example scratchpad use.** Figure 7 shows an example scenario where \( \text{spld} \) 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 12) 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.

B. 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 (PMMC) 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 II. We disable the PosMap Lookaside Buffer (PLB) in Frecursive ORAM to avoid leakage through the total number of ORAM accesses.

C. 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.

VI. Evaluation

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

TABLE II: 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>LUT</th>
<th>FFs</th>
<th>LUT-Mem</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Estimate (% Utilization)</td>
<td>169472</td>
<td>51870</td>
<td>81112</td>
<td>566.5</td>
</tr>
<tr>
<td>HOP Estimate (% Utilization)</td>
<td>103462</td>
<td>39803</td>
<td>38725</td>
<td>437</td>
</tr>
<tr>
<td>(HOP – ORAM) Estimate (% Utilization)</td>
<td>221041</td>
<td>81410</td>
<td>81126</td>
<td>566.5</td>
</tr>
</tbody>
</table>

A. 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 an $A^N M$ schedule (Section III-C).

For the programs we evaluate (except bzip2, c.f., Section VI-D), 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 V-B, 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 III-C, 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).

B. Area Results

We synthesized, placed and routed HOP on a Xilinx Virtex V7485t FPGA for parameters described in Section V. HOP operates at 79.3 MHz on this FPGA. The resource allocation and utilization figures are mentioned in Table II. The first three rows represent the total estimate, estimate for HOP (i.e. excluding RISC-V processor, 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.


**C. Main Results**

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 III-B only); the second bar only uses an $A^N M$ schedule without a scratchpad (adds Section III-C); and the third bar is our final scheme that uses a scratchpad and the $A^N M$ schedule (adds Section III-D). 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 V-A. 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 III-C.

**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, heapop 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 $40 \times$ more expensive than an insecure access. Second, dummy accesses to adhere to a schedule: As shown in Section III-C, the performance overhead due to dummy accesses $\leq 2 \times$. For programs such as bwt-rle, HOP has a slowdown as low as $8 \times$. This is primarily due to the reduction in ORAM accesses by maintaining a small working set in the scratchpad.

**D. 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 $106 \times$ speedup over the baseline scheme and $17 \times$ slowdown over the insecure version. For the second input, HOP achieves $234 \times$ speedup over the baseline and $8 \times$ slowdown over the insecure version. Thus, the gains and slowdowns we see from the prior studies extend to this more sophisticated benchmark.

**E. 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.

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 III 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 $18 \times c \times \log c$ 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. 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 V.

<table>
<thead>
<tr>
<th>Function</th>
<th>FHE [15], [17] Hardware [27]</th>
</tr>
</thead>
<tbody>
<tr>
<td>findmax</td>
<td>On+Off</td>
</tr>
<tr>
<td>1 s × 10^9</td>
<td>2 × 10^9</td>
</tr>
<tr>
<td>binsearch</td>
<td>4 × 10^9</td>
</tr>
</tbody>
</table>

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

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 (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.

2) Comparison with iO [37]: We compare HOP with an implementation of indistinguishability obfuscation (iO) that does not assume a trusted hardware token. Note that while VBB obfuscation is not achievable in general, iO 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.

3) Comparison with GhostRider [40]: Recall from Section II 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 (sum, findmax, hist), GhostRider’s performance is similar to that of an insecure processor.

F. Time for Context Switch

Since it was not required for our performance evaluation, we have not yet implemented context switching (Section III-E) 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 ∼ 800 KB (Section V). Assuming a DRAM bandwidth of 10 GB/s and a matching encryption bandwidth, it would take ∼ 160µs 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.

VII. PRACTICAL DEPLOYMENT AND APPLICATIONS

In this section, we present practical deployment considerations and potential applications for a HOP processor.

Potentials involved in the system. In a practical deployment, there would be three parties involved in this system: The sender is a software provider (e.g., Microsoft), the receiver is the end user and the manufacturer is a hardware company (e.g., Intel, TSMC). Software providers are incentivized to use this framework to hide the IP of their proprietary programs so as to sell those programs to customers without being pirated. Hardware manufacturers are incentivized to provide security in order to retain the software providers as customers (e.g., Intel SGX was initially envisioned as a buy-in service).

Potential applications. The focus of this paper is to build hardware that satisfies the definition of VBB obfuscation. Thus, our model assumes obfuscation of ‘batch programs’ – those which take inputs, and compute non-interactively until a result is produced. Some examples of such programs are compilers, compression algorithms, machine learning algorithms, etc. In our setting, the program itself should contain some sensitive IP (to warrant obfuscation). Given the pervasive nature of batch programs, we see HOP being applicable in both commercial and military settings. We note that even the military outsources its fabrication (and therefore its trust) to external foundries (e.g., global foundries handles runs for the NSA).

Beyond batch programs, it is possible to support streaming applications with little change to the model. In particular, while HOP runs, it can accept streams of public data (e.g., a video feed) in a similar fashion to Stream Ascend [57]. Importantly, this has no impact on security, as long as HOP doesn’t change its observable behavior given the data in the stream and the program accepts this data at fixed intervals of time.

---

6When 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 ~ 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.
VIII. 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 $8 \times \sim 7\times$ 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.

Acknowledgment

We would like to thank anonymous reviewers for their insightful feedback. This work is supported in part by NSF grants CNS-1314857, CNS-1453634, CNS-1518765, CNS-1514261, an ONR-YIP Award, a Packard Fellowship, a Sloan Fellowship, a Google Ph.D. Fellowship, Google Faculty Research Awards, a DARPA Brandeis grant, a DARPA Safeware grant and a VMware Research Award. This work was done in part while a subset of the authors were visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant CNS-1523467.

References

by a continuous run of memory instructions. Denote the $i$-th epoch as $A^iM^p$. For example, the program

\begin{equation*}
A A A M A A M M M M
\end{equation*}

has 2 epochs, with $n_1 = 4, p_1 = 1, n_2 = 2, p_2 = 3$.

Without loss of generality, we align the start of each epoch with the beginning of an $A^3M$ schedule. Given our choice of $N$, we examine the number of processor cycles spent doing dummy operations in each epoch. For the rest of the analysis, we abbreviate $|M| = \text{ORAM latency}$ and $|A| = \text{Arithmetic latency}$.

Consider the start of epoch $i$ (i.e., the first $A$ instruction). To progress from the start of the epoch to the first $M$ instruction (excluded in the epoch), we perform $|A| + N\lfloor \frac{n_i}{N} \rfloor + \lfloor A\lfloor n_i \mod N \rfloor \rfloor$ real cycles and $|M| + \lfloor \frac{n_i}{N} \rfloor + \lfloor A\lfloor N - (n_i \mod N) \rfloor \rfloor$ dummy cycles worth of work. To progress from the first $M$ instruction (including to the end of the epoch, we perform $|M| \times p_i$ real cycles and $|A| \times N \times (p_i - 1)$ dummy cycles worth of work. Note that by our definitions of epochs, we have that $p_i \geq 1$.

Also note that $|M| = |A| \times N$ by our choice of $N$. Combining these two time periods, we spend $\sum |M| \times \left( \lfloor \frac{n_i}{N} \rfloor + p_i - 1 \right) + \lfloor A\lfloor N - (n_i \mod N) \rfloor \rfloor$ dummy cycles worth of work.

\section*{Appendix B

\textbf{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 obfuscate 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 $sk_{\text{enc}}$ of a public-key CCA secure encryption scheme (with public key $pk_{\text{enc}}$) along with a verification key $vk_{\text{sig}}$ of a signature scheme (with signing key $sk_{\text{sig}}$). The signing key $sk_{\text{sig}}$ 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 ciphertexts 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 $(sk_{\text{sig}}, vk_{\text{sig}})$, $S_1$ will obtain a signature of $vk_{\text{sig}}$ from the trusted certificate authority (denote this signature by $\sigma$ and note that this signature will verify under the verification key $vk_{\text{sig}}$). Now, $S_1$ will encrypt $P_1$ with $pk_{\text{enc}}$ and authenticate all ciphertexts with $sk_{\text{sig}}$ and provide these ciphertexts along with $\sigma$ to the receiver. The receiver will feed in encrypted ciphertexts along with $\sigma$ to the token. The token, when decrypting ciphertexts, will first check the validity of $vk_{\text{sig}}$ by verifying $\sigma$ and the signatures of all the ciphertexts. If all the checks pass, the token will decrypt the ciphertexts using $sk_{\text{enc}}$. When encrypting state to be sent back to the receiver, the token will encrypt it with $pk_{\text{enc}}$ and sign it with $sk_{\text{sig}}$. This will mimic the symmetric key CCA secure authenticated encryption scheme that we use in our single sender/receiver scheme.