# BliMe: Verifiably Secure Outsourced Computation with Hardware-Enforced Taint Tracking

Hossam ElAtali
*University of Waterloo*
hossam.elatali@uwaterloo.ca

Lachlan J. Gunn
*Aalto University*
lachlan@gunn.ee

Hans Liljestrand
*University of Waterloo*
hans@liljestrand.dev

N. Asokan
*University of Waterloo*
*Aalto University*
asokan@acm.org

***Abstract*—Outsourced computing is widely used today. However, current approaches for protecting client data in outsourced computing fall short: use of cryptographic techniques like fully-homomorphic encryption incurs substantial costs, whereas use of hardware-assisted trusted execution environments has been shown to be vulnerable to run-time and side-channel attacks.**

**We present Blinded Memory (BliMe), an architecture to realize efficient and secure outsourced computation. BliMe consists of a novel and minimal set of instruction set architecture (ISA) extensions implementing a taint-tracking policy to ensure the confidentiality of client data even in the presence of server vulnerabilities. To secure outsourced computation, the BliMe extensions can be used together with an attestable, fixed-function hardware security module (HSM) and an encryption engine that provides atomic decrypt-and-taint and encrypt-and-untaint operations. Clients rely on remote attestation and key agreement with the HSM to ensure that their data can be transferred securely to and from the encryption engine and will always be protected by BliMe's taint-tracking policy while at the server.**

**We provide an RTL implementation BliMe-BOOM based on the BOOM RISC-V core. BliMe-BOOM requires no reduction in clock frequency relative to unmodified BOOM, and has minimal power ($<1.5\%$) and FPGA resource ($\leq 9.0\%$) overheads. Various implementations of BliMe incur only moderate performance overhead (8–25%). We also provide a machine-checked security proof of a simplified model ISA with BliMe extensions.**

## I. Introduction

Outsourced computation has become ubiquitous. While cost-effective, outsourced computation introduces confidentiality concerns because the clients' sensitive data must be sent to the cloud service provider (CSP)'s servers for processing by some (possibly third-party) application software. Malicious CSPs or software can misuse client data. Furthermore, even if the CSP is trusted, other malicious actors may compromise the sensitive data through run-time attacks or side-channel leakage.

One solution to this problem is to use cryptographic techniques such as fully-homomorphic encryption (FHE) which allow arbitrary computation on encrypted data. With FHE, the server receives and processes only encrypted data. However, FHE incurs very large performance overheads, orders of magnitude worse than processing the plaintext directly [62]. An alternative is to use hardware-assisted security mechanisms like trusted execution environments (TEEs) present in modern central processing unit (CPU) architectures, which can potentially ensure security while maintaining performance. A TEE provides isolation for each client from the server operating system (OS) as well as from other clients. But bugs in server software can lead to run-time attacks. Furthermore, most TEE implementations offer only limited resistance to side-channel attacks [14], [15], [45]. Therefore, the need remains for an efficient and effective approach to protect outsourced computation even in the presence of software vulnerabilities and side channels.

Our goal is to address this need via minimal hardware changes. We design a minimal set of extensions for the RISC-V instruction set architecture (ISA) to preserve data confidentiality even in the presence of software vulnerabilities or side channels. The extensions and the accompanying attestation architecture, which together we call Blinded Memory (BliMe), allow a client to send conventionally encrypted data to a remote server, so that the CPU can decrypt and process the data without allowing it or any data derived from it to be exfiltrated from the system. Computation results are returned only after encryption with the client's key.

We do this by having the CPU enforce a taint-tracking policy preventing client data from being exported from the system. Prior work on hardware-enforced taint tracking [67] provide an untaint instruction to extract results, implicitly assuming that software invoking this instruction is trusted, making them vulnerable to run-time or speculative execution attacks [53]. In contrast, BliMe uses a small attestable, fixed-function hardware security module (HSM) and an encryption engine to facilitate secure import and export of data between clients and servers; decryption on server-side always results in tainted plaintext. This allows BliMe to provide its security guarantees to *multiple independent clients*, who *do not need to trust server software*, including the OS. Consequently, BliMe protects not just against side channels, but even against malware and run-time attacks that allow an attacker to execute arbitrary server software. Our contributions are:

- BliMe, an architecture with a set of taint tracking ISA extensions for preventing exfiltration of sensitive data

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24105
www.ndss-symposium.org

(Section IV).

- BliMe-BOOM, an RTL implementation of the BliMe ISA extensions, incorporated into the speculative out-of-order RISC-V core BOOM (Section V).
- A machine-checked proof of the dataflow security of the BliMe taint-tracking policy applied to a simplified model ISA (Section VI).
- A performance evaluation of BliMe-BOOM showing minimal run-time, power and area overheads (Section VII).

## II. Background

### A. Trusted Execution Environments

TEEs isolate trusted applications (TAs)—programs within TEEs—from software outside the TEE as well as from other TAs. In addition to TA isolation, TEEs also provide *remote attestation* to assure clients that the code and configuration of server-side components are what they expect. TEEs are already present in several x86, Arm and RISC-V CPUs, and are available to clients through CSPs such as Amazon, Google, and Microsoft.

TEEs contain a root of trust for attestation, typically in the form of a unique attestation key embedded in the hardware at the time of manufacture. The remote attestation protocol first authenticates the hardware by having the server prove that its TEE possesses this key, which is certified by the device manufacturer. After authentication, remote attestation is provided by "measuring" the system: the code, configuration, and state of the system are checked by the TEE to assure the client that they are as expected.

Despite the claimed isolation guarantees, TEEs still suffer from certain vulnerabilities. Isolation only prevents malicious processes from naively accessing a TA's data through direct memory access. Remote attestation and code attestation only assure the clients that the system is set up as expected and that the code used to process the data is unmodified. Software bugs, present even in attested code, can lead to run-time attacks that circumvent client isolation, giving adversaries direct access to sensitive data. Despite many defenses, memory vulnerabilities and run-time attacks are still pervasive.

### B. Side-channel leakage

Even if the software is bug-free, side-channel leakage can occur due to vulnerabilities or data-dependent behavior in the underlying hardware. Side channels are observable outputs of the system that are not part of the system's intended outputs. Prominent examples of CPU side channels are execution time, memory access patterns, observable microarchitectural state (such as the state of shared caches, branch predictors and performance counters), voltage and electromagnetic radiation [43], [32], [16], [37], [35], [29]. Side-channel leakage can occur when an adversary is able to infer information about the sensitive data by observing the system while the data is processed. For example, if a conditional branch instruction depends on a sensitive value and the execution time of each branch is different, an adversary can infer some information about the sensitive value by monitoring the time it takes to complete the branch. Another example is when a sensitive value is used to index an array in memory; the memory access pattern, which is the sensitive address in this case, can change the observable state of a shared cache, or result in an observable request on the main memory bus.

Modern CPUs employ a variety of performance optimizations in the form of speculation and out-of-order execution that amplify the leakage caused by data-dependent behavior [36], [31], [52], [61], [46], [61], [18], [10], [34], [12], [50], [69]. This behavior is largely transparent to software developers, making it difficult to detect when benign-looking code can cause side-channel leakage.

## III. Problem Description

### A. Usage scenario

The scenario we target is where several clients send data to a remote server for outsourced computation. Each client starts a session with the server, sends its data, and the server invokes some *potentially untrusted, third-party* software ("server software") to perform computation on the data (possibly in combination with the server's own data). Once computation completes, the results are sent to the client. Data import/export and computation can repeat multiple times per session. Data exchange between clients and servers is secured using authenticated encryption. Multiple clients may connect to the server simultaneously, leading to multiple parallel sessions.

Execution of server software that can leak any information about client data must be prohibited, even when attackers can run malware on the server, exploit vulnerabilities in the server software, or use side channels to extract data. In other words, sensitive client data must not flow to any observable output, nor to any other client.

### B. Goals and objectives

We now identify design requirements. The first relates to security.

**SR—Confidentiality**: *When a client provides sensitive input data to the server, no party other than that same client can infer anything about this data, other than its length.*

Malware running on the server may attempt to gain access to the data, or the software processing the sensitive data may itself be malicious, but must not be allowed to reveal sensitive data outside the system, or to anyone other than the original client.

The next requirement relates to performance.

**PR—Fast execution**: *The design will not significantly reduce the performance of software accepted by the CPU, compared to running the same software on a similar processor without such protections.*

It is important to ensure that any solution does not excessively degrade performance. Elimination of side channels may prevent certain optimizations, resulting in some overhead, but some high-performance security-critical software has already been hand-written in assembly to eliminate side channels,

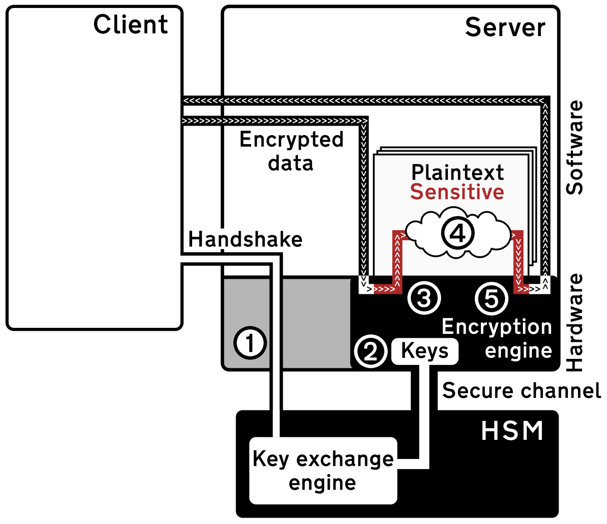

Fig. 1. BliMe Overview. The client completes a cryptographic handshake with HSM ①, which stores the resulting shared key in a protected register for use by the encryption engine ②. The client encrypts its secret data, which the server software decrypts with the aid of the encryption engine, while atomically marking the decrypted data as blinded ③. The server software can then perform the requested computation on the resulting blinded data in a verifiably leakage-resistant manner ④, and encrypt the output using the encryption engine ⑤ for the client.

and solutions that significantly reduce its performance may prove unsuitable in applications that make heavy use of such software.

The final requirement relates to backwards-compatibility.

**CR—Backwards compatibility**: *Software that does not leak sensitive data, by covert channels or otherwise, will run successfully.*

A large portion of server software does not process sensitive data. It is important from a practical point that existing software can run on the new hardware: if this is not the case, then a new software stack will be needed, greatly limiting utility.

Moreover, there is also software that handles sensitive data but that already does so safely, such as side-channel-resistant cryptographic software. It is equally desirable that this secure and well-tested software will continue to run on our hardware.

## IV. Design

### A. System overview

Figure 1 shows an overview of BliMe. The server's software, including the OS, runs on top of a CPU that contains 1) the BliMe extensions, which enforce a taint-tracking policy on all software running on the CPU (Section IV-D), and 2) a BliMe encryption engine, used for data import and export (Sections IV-C2 and IV-C4). A HSM is used to perform remote attestation, assuring the client that the server uses BliMe. The HSM is a separate fixed-function component sharing few resources with the CPU, reducing its exposure to side-channel attacks. The client knows how to verify the root of trust for attestation embedded within the HSM, which contains a key exchange engine responsible for negotiating a session key between with the client. The HSM provides this session key securely to the encryption engine without exposing it to the server software.

### B. Adversary model

We suppose that the server hardware, including the BliMe extensions, encryption engine, and the HSM, is implemented correctly.

The adversary has control over all server software, including the OS, and can make it behave as it sees fit, including making inferences based on side channel information such as memory access patterns and instruction traces.

Attacks against the hardware itself are currently out-of-scope; the client assumes that the attacker cannot use physical means to make the hardware act differently from its specification. We discuss this in Section IX-E. Side-channel attacks that require physical access (e.g., differential power analysis) are also out-of-scope.

### C. Protocol

In this section, we outline the steps needed to perform safe outsourced computation using BliMe.

*1) Remote attestation and key agreement:* Before sending any data to the server, a client first performs a handshake (Figure 1-①) with the HSM, which consists of remote attestation and agreement on a session key. Remote attestation is provided by the HSM to assure the client that the assumptions made in the adversary model hold. It attests two properties. First, the HSM attests that it is genuine using the root-of-trust embedded within it at manufacture, as described in Section II-A. Second, the HSM attests to the client that it is embedded in server hardware incorporating BliMe.

At the end of remote attestation, the client and HSM agree on a client-specific session key. The HSM stores this key inside the encryption engine using a secure channel (Figure 1-②), along with a unique blindedness tag identifying the client from which data was sourced; this is most simply implemented cryptographically using a sealing key shared by the two components. The encryption engine uses the session-specific key in two atomic functions that it exposes to the server software: *data import* and *data export.*

*2) Data import:* Once the handshake is complete, the client locally encrypts its data using the session key, and sends the resulting ciphertext to the server. The server software calls the encryption engine's data import function on the ciphertext. The encryption engine then *atomically* decrypts the ciphertext using the sealed session key, and taints the resulting plaintext by *marking* it as *"blinded"* (Figure 1-③). This is done by setting a *blindedness* tag attached to the data in registers and memory. The blindedness tag is an $n$-bit integer, which takes the value zero when its associated data is not blinded. Other values indicate that the data is blinded, and from which of $2^n - 1$ clients the data is sourced. In the minimal case where $n = 1$, this tag is a single bit that indicates whether or not the data is blinded.

Data import atomically decrypts data using the session key and marks it as *blinded*.

*3) Safe computation:* Now, the server software can perform the requested computation on the blinded plaintext data (Figure 1-④). The BliMe CPU extensions apply a taint-tracking policy, limiting the operations that can be done on blinded data; this prevents the server software from directly exfiltrating the data or even leaking it through side channels. Any operations forbidden by the taint-tracking policy cause the server to fault. The policy ensures that the final results and any intermediate results derived from the data are also blinded. More details are in Section IV-D.

*4) Data export:* Once the computation is complete, the server software calls the encryption engine's data export function to atomically encrypt the blinded results and mark the ciphertext as non-blinded (Figure 1-⑤), which is done by zeroing the blindedness tag. The server software can then send the ciphertext back to the client, who can decrypt it. The encryption engine ensures that blinded data is only encrypted with the session key corresponding to its blindedness tag. As the adversary controls the data to be encrypted, this encryption must be secure against adaptive chosen-plaintext attacks in order to prevent the adversary from using this ability to identify the ciphertexts corresponding to particular plaintexts.

Data export atomically encrypts data using the session key and marks it as *non-blinded*.

## D. Taint-tracking policy

We use taint tracking to *prevent sensitive data from flowing to observable outputs*. First, we define which parts of the system can be tainted. We split the system state into two types:

- *Blindable* state consists of the values (not addresses) of lines in the cache, values in registers except the program counter, and values in main memory, as well as all busses and queues used to transfer these values. Each of these is extended with a blindedness tag.
- *Visible* state consists of information that may be exposed outside the system, and must therefore never contain sensitive data. It includes *all microarchitectural state that does not have a blindedness tag associated with it*, e.g., the program counter, addresses of lines in the caches, branch predictor state, and performance counters. As the program counter encapsulates control flow, making it part of visible state forbids blinded data from affecting control flow.

We then define the list of observable outputs we consider in BliMe: visible state, non-blinded blindable state, the addresses of memory operations sent to main memory, the execution time of an instruction or set of instructions, and fault signals. Note that once blindable state becomes blinded, it is no longer observable. We exclude outputs that require physical access to be observed, such as voltage and electromagnetic radiation.

The taint-tracking policy is defined as follows (and as shown in Table I). An instruction with a blinded input (i.e., which takes at least one input with a non-zero blindedness tag) yields a blinded value for each output that depends on a blinded input, with the same blindedness tag as the input. An instruction that receives blinded data from multiple sources will raise a fault. If an instruction attempts to affect any observable output except non-blinded blindable state in a manner that depends on the value of a blinded input, a fault is raised, since this can otherwise be used to exfiltrate sensitive data. This effectively means that the program cannot use a blinded value as the address of a jump, branch or memory access, or use instructions whose completion time or fault status depends on a blinded value. For example, BliMe prevents

- *Cache-timing side-channel attacks* by prohibiting blinded values from being used as addresses for loads/stores, and
- *Timing attacks* by forbidding blinded-value-dependent control flow (prohibiting both altering PC based on blinded values, and instructions of variable duration from using blinded values). Measuring program execution time will reveal nothing about the blinded values; the adversary will obtain the same information as if they ran the same program over a blinded array of zeroes of the same length.
- *Transient execution attacks* by preventing any blinded values from being used in speculation decisions. Speculatively executed instructions can still use blinded values (and maintain their blindedness tags throughout this transient execution), but the result of any prediction *decision* (e.g., whether a branch is taken, or what the next return address might be) cannot rely on blinded values. This same concept applies to all data-dependent hardware optimizations; blinded values cannot be used to *decide* on whether or how an optimization is applied.

The blindedness of any given value is not sensitive; this means that it is safe for the ISA to include instructions that query whether or not a given value is blinded.

## E. BliMe-compliant software

BliMe's restrictions on application software are the same as those required of anyone developing secure side-channel-resistant code (even without BliMe hardware modifications): they must adhere to constant-time coding principles [5], including data-oblivious control flow and memory access, and must not explicitly exfiltrate sensitive data outside the system. An example of such practices being used today can be found in the development of cryptographic libraries, the compatibility of which we show in Section VIII. Concretely, for software to be BliMe-compliant, it must not attempt to:

1) use blinded data in any control-flow decisions,
2) use blinded data as the target address for any jump or branch instructions,
3) use blinded data as the address for any memory operations, including using blinded data as an offset or index,
4) use blinded data as an operand for an instruction whose execution time depends on the value of that operand (e.g., variable-time division instruction),

| Instruction | Tag (Ø=unblinded, ?="don't care") | | Decision | Result tag |
|---|---|---|---|---|
| Arithmetic/Logic[†] | If # of cycles depends on value of any blinded operand | | Fault | - |
| | Otherwise: | | | |
| | **Op1** | **Op2** | | |
| | Ø | Ø | Propagate | Ø |
| | $a$ | Ø | Propagate | $a$ |
| | $a$ | $a$ | Propagate | $a$ |
| | $a$ | $b$ | Fault | - |
| Branching | **Addr** | **Condition ops** | | |
| | Ø | Ø | Propagate | Ø |
| | $a$ | ? | Fault | - |
| | ? | $a$ | Fault | - |
| Load | **Addr** | **Data in memory** | | |
| | Ø | Ø | Propagate | Ø |
| | $a$ | ? | Fault | - |
| | Ø | $a$ | Propagate | $a$ |
| Store | **Addr** | **Data in register** | | |
| | Ø | Ø | Propagate | Ø |
| | $a$ | ? | Fault | - |
| | Ø | $a$ | Propagate | $a$ |

[†] See Section IX-D for a discussion on data-dependent faults.

TABLE I. BLIME TAINT-TRACKING POLICY RULES FOR ALL INSTRUCTION TYPES. TAGS $a$ AND $b$ ARE ARBITRARY BUT DIFFERENT. SWAPPING THEM PRODUCES AN EQUIVALENT RESULT.

5) mix blinded data belonging to separate security domains, i.e., with different blindedness tags, and
6) write blinded data to any peripherals, e.g., displays, network devices, disk drives.

Note that BliMe does *not* require software developers to be aware of how the CPU behaves speculatively. BliMe ensures that all blinded data cannot be leaked by hardware optimizations, including speculation (Section IV-D).

These requirements do not limit what can be computed. A much more restrictive historical CPU, Zuse's Z3, lacks conditional branching and indirect addressing but can simulate any finite-tape Turing machine [49]. BliMe allows conditional branching and indirect addressing on non-sensitive data as well as all of Z3's functionality, making it as powerful as but more efficient than Z3. Therefore, BliMe can simulate any finite-tape Turing machine.

## V. Implementation

In this section, we first present the common architectural changes required to implement BliMe (Section V-A), and then describe how these were applied to the out-of-order RISC-V BOOM core [70] to obtain BliMe-BOOM (Section V-B).

The implementation covers the hardware needed for safe computation (Section IV-C3), including taint tracking and enforcement of the security policy (Section IV-D). We do not include the HSM in the implementation. Components with similar functionality already exist, such as Google's Titan-M chip [39] or Apple's Secure Enclave Processor (SEP) [2]; the HSM is configured to perform the attested handshake and provide correct client IDs, and the system integrator provides the secure channel between HSM and encryption engine in the form of a shared key embedded in each component.

### *A. Architectural changes*

The architectural changes needed to implement BliMe can be grouped into two main categories: taint tracking, and handling of policy violations.

**Taint-tracking** is performed on registers and memory. Due to the load-store architecture of RISC-V, these can be handled separately.

**Registers and ALU operations.** For each register, we maintain a blindedness tag. Any ALU operation that reads from a blinded register, blinds its output register by default. However, this is a conservative approximation, and implementations can make exceptions in order to more accurately model instruction dataflows. For example, if a register is XORed with itself, the result is not blinded because the result is always zero, irrespective of the input value. Similar exceptions can be used for other situations in which an instruction takes blinded inputs but its output does not depend on said blinded inputs.

**Memory.** Blindedness is tracked in using a tagged-memory approach, with a blindedness tag being attached to each physical address, in both main memory and in each cache. Whenever a value is stored into memory from a register, the memory bytes inherit the blindedness of the register. The reverse holds for memory reads. To reduce overheads, multiple consecutive bytes in memory, forming a *tag granule*, can share the same tag. For tag granule sizes larger than a single byte, we introduce an additional BliMe policy rule to prevent mixing blinded data belonging to separate security domains: if a store instruction attempts a *partial* write of blinded data (e.g., a byte) with tag $a$ to a granule in memory (e.g., of size 8 bytes) with tag $b$, the instruction must fault.

**New Instructions.** We introduce two new instructions, `blnd` and `rblnd`, that correspond to the data import and export operations, respectively (Sections IV-C2 and IV-C4). The instructions encrypt and decrypt data with a supplied key, writing the result into memory in unblinded or blinded form respectively.

**Handling violations** is needed in four situations:

1) Attempting to write blinded values to the `PC` register: Jumps and conditional branches relying on blinded registers, either as a jump destination address or as part of a conditional check, are forbidden.
2) Attempting to use blinded values in an instruction whose execution time depends on the blinded input

value.

3) Attempting to read from or write to memory using a blinded value as an address: This occurs when a load or store uses a blinded register either as the address base or offset.
4) Attempting to write blinded data to an "unblindable" memory location. This allows a system to specify whether certain memory-mapped peripherals have access to blinded data.

Any of the above causes an illegal instruction fault.

We implemented the architectural changes in Spike [48], a C++ RISC-V ISA simulator, to enable quick testing of our extensions.

## B. BliMe-BOOM

To demonstrate BliMe on a complex and realistic computation platform that relies heavily on speculation to achieve high performance, we implemented BliMe in RTL on the BOOM [70] core and Chipyard [1] system-on-chip (SoC). BliMe taint-propagation rules are also enforced during any speculative execution thereby preventing sensitive values from being leaked even by Spectre-type attacks [31].

We implemented two variations of BliMe with different tag configurations: BliMe-BOOM-1 with single-bit blindedness tags and byte-level granularity, and BliMe-BOOM-8 with eight-bit blindedness tags and 8-byte (i.e., word-level) granularity.

The BOOM core is written in Chisel, which is a hardware construction language embedded in Scala [6]. Chisel defines common built-in data types, such as unsigned integers (UInt) and Boolean values (Bool), used to attach semantic meaning to bits in digital circuits. Chisel also allows user-defined data types to be created by composition and inheritance. Taking advantage of this feature, we create a *Blinded* data type as a composition of a blindedness tag and a variable-length vector of bits, and use this type throughout the design. As Scala (and, by extension, Chisel) is strongly-typed, this allows the compiler to ensure that blinded values cannot be separated from their blindedness tags except by code that is aware of the *Blinded* type. It also allowed us to identify *all locations in the code* where blinded data can propagate and prevent any unintended leakage caused by hardware optimizations such as speculation.

*1) Registers:* Register files are modified to use the *Blinded* type, thereby storing a blindedness tag alongside the register contents.

*2) Main Memory & Caches:* A region of memory is allocated for the storage of blindedness tags, and made inaccessible to software. We expand the L1 data and L2 caches to include blindedness tags alongside each granule of data. The L1 instruction cache does not need blindedness tags; blinded values are not allowed to be executed, so whenever a blinded value is read into the cache, it is replaced with a zero, and its 'valid instruction' metadata bits are unset, making it inaccessible. The L2 cache includes logic to translate misses into *two* operations to main memory: one for the actual data, and one for the blindedness tags.

It is also possible to optimize the design for the specific case where the blindedness tags are small relative to their associated data, with carefully-designed logic reducing the ratio of blindedness tag requests to data requests, thereby reducing overhead. However, a limitation in Chipyard's TileLink implementation allows data to be read only at an 8-word granularity, yielding a 64-bit effective minimum tag size. Therefore, even for 8-bit tags, the memory controller ties up the bus for seven additional cycles to transmit unneeded words. This is not a limitation of BliMe, but of Chipyard. We forego implementing the extensive modifications required to the memory controller and instead investigate the effect of this optimization using gem5 [38] in Section VII-B.

*3) Pipeline Stages:* In BOOM, instructions are decoded in the instruction decoder stage into micro-ops, which are stored in the reorder buffer until they are issued to an execution unit and committed. Execution units contain different functional units that can perform specific operations. We modified all functional units to either propagate blindedness (e.g., addition of blinded values), or fault when the operation is unsafe to perform with blinded operands (e.g. memory address generation).

The functional unit responsible for memory address generation faults when blinded operands are used to generate a virtual address. Furthermore, if the page table walker finds a blinded page table entry (PTE), it zeroes it before storing it in the translation-lookaside buffer (TLB). This ensures that virtual-to-physical address translation by the load-store unit cannot be used to load from a blinded physical address. Any load or store attempting to use a blinded physical address directly will fault.

*4) Encryption engine:* The encryption engine, which performs atomic blind-and-decrypt and encrypt-and-unblind operations using the ChaCha20 stream cipher, is built as a RoCC accelerator [3, p. 3] containing a pipelined ChaCha20 implementation with a 19-cycle latency. This allows this functionality to be accessed using a set of instruction opcodes reserved for accelerators, and allows it to be used by an unmodified C compiler using RISC-V intrinsics.

*5) Speculation:* Speculatively executed instructions are subject to the same checks as non-speculative instructions regarding blinded operands. Branch prediction can also never depend on blinded data. This is done by blocking all blinded data flows into the prediction logic. We 1) zero out any blinded instruction fetched into the instruction cache, 2) prevent any feedback from the execution units to the prediction logic regarding faulting branch decisions based on blinded data. By ensuring that both the speculation decision *and* the speculative execution do not leak sensitive data, we block all speculative side channels. This is in a similar vein to Speculative Taint Tracking (STT) [68] but, unlike STT, we do not untaint after the instructions leave the speculative state and the security policy on the data remains in effect.

Note that, for BliMe-BOOM, only non-constant-time operations, which always fault or are rolled back, affect speculation. Constant-time code, with no secret-data-dependent branching or memory accesses, speculates normally *without overhead*. This is because BOOM does not have any other data-dependent speculation (e.g., data-dependent prefetchers).

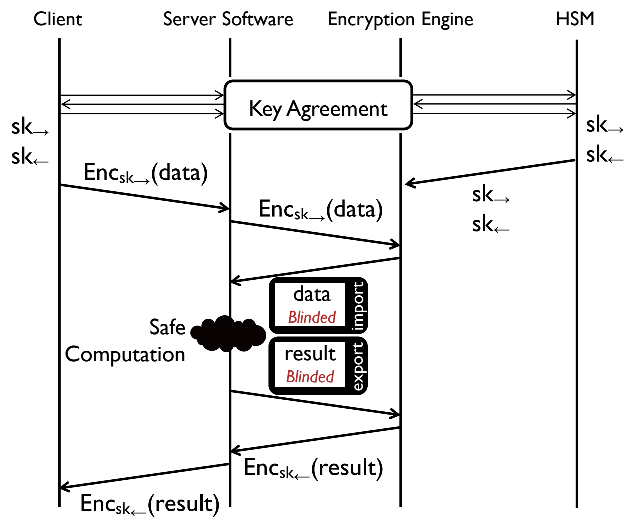

Fig. 2. Protocol summary. A key agreement protocol is used to obtain secret keys, shared by the client and HSM. The HSM securely transports these keys to the encryption engine, which atomically decrypts and blinds the client data, performs safe computation on it, then atomically encrypts and blinds the client data before it is returned to the client.

For microarchitectures with such speculation, it must be disabled (only) on blinded data.

## VI. Security Evaluation

### A. Protocol

Computation using BliMe takes the form of the protocol shown in Figure 2. This is equivalent to the protocol described in Section IV-C. The attacker obtains only encrypted client data, the encryption of an arbitrary computation on the plaintext data, along with leakage from the computation by the server code, shown as a cloud in Figure 2.

If the key agreement protocol and secure channel protocols are secure against active attackers, then the messages provide the attacker with no information on either the session keys, input data from the client, or the result of the computation. Thus, the attacker gains information on the client data only if the computational leakage (during the Safe Computation phase in Figure 2) reveals some information. It is therefore sufficient to show that the leakage is independent of the decrypted plaintext, in order to meet requirement SR.

### B. Safe computation

The decrypted plaintext, shown in Figure 2, is always marked as blinded. Therefore, it is sufficient to prove that the safe computation functionality described in Section IV-C3 does not leak any information on any state marked as blinded.

We do this by constructing a model ISA using the F* programming language [56], and proving that its visible and non-blinded states are independent of the blinded values in memory: that is to say, no changes to the blinded values containing sensitive data, after a computational step, result in any change to the non-blinded values or blindedness bits that can be observed by the attacker.

Our approach is as follows:

1) Define a low-level model of the server in terms of registers, memory cells, cache line allocations, and arbitrary transitions between states, and a security definition with respect to this model that meets the requirements of Section VI-A.
2) Define a more detailed architectural model that expresses instructions in terms of register and memory reads/writes, and a security definition that implies the low-level security definition from step 1.
3) Define a minimal instruction architecture that implements standard arithmetic operations with special-case taint propagation rules from Section IV-D, and prove that it satisfies the security definitions from steps 1 and 2.

*1) Definitions:* We begin by defining equivalence relations on values that may be blinded. We define a recursive type `multiBlinded`[1] that represents a blindable value:

```
type blindedness_domain =
    x:FStar.UInt8.t{~(x = FStar.UInt8.zero)}
type multiBlinded (#t:Type) =
  | MultiClear: v:t
      -> multiBlinded #t
  | MultiBlinded: v:t
      -> d:blindedness_domain
      -> multiBlinded #t
```

Then, a blinded value $v$ with tag $d$ is represented by the value `MultiBlinded` $v$ $d$, and a non-blinded value $v$ by `MultiClear` $v$.

We say that two such values $a$ and $b$ are equivalent, denoted $a \equiv b$, if they are both `MultiBlinded` with the same tag, or if they are both `MultiClear` and have equal values.

We define equivalence of lists of `multiBlinded` values similarly: two lists $\ell_1$ and $\ell_2$ are equivalent, denoted $\ell_1 \overset{\text{list}}{\equiv} \ell_2$, if their lengths are equal, and each of their values is equivalent.

*2) Low-level system model:* The system is modelled in Cpu.fst by a system state type containing the following data:

- the program counter (`pc`), containing a 64-bit unsigned integer pointing into memory,
- an array of register values, each containing a blindable 64-bit unsigned integer,
- an array of memory values, each containing a blindable 64-bit unsigned integer, and
- an array of cache line assignments, each containing a 64-bit unsigned integer representing the address of the corresponding value in the cache.

We then define an equivalence relation $\overset{\text{state}}{\equiv}$ (`equiv_system` in the model) on the system states $\mathcal{S}$, such that two system states are equivalent if they have equal program counters and cache line assignments, their registers and memory have equal blindedness tags, and their non-blinded values are equal.

[1]We include links to specific modules and theorems where they are mentioned. The full model is available at https://blinded-computation.github.io/blime-model/.

We model the execution of instructions using a single-cycle fetch-execute model (`step` in the model), with each instruction completing in a single cycle[2]. An instruction $I \in \mathcal{I}$ is loaded from memory at the program counter address; if `pc` points to an instruction marked as blinded, then it jumps to a fault handler at address $0$. Otherwise, the state of the processor is transformed according to an instruction-dependent execution mapping $X : \mathcal{I} \times \mathcal{S} \to \mathcal{S}$. We denote a single processor step

$$P_X(s) = \begin{cases} X(s.\text{MEMORY}[s.\text{PC}], s), & \text{if } s.\text{PC points to} \\ & \text{non-blinded memory,} \\ s \text{ with s.PC} = 0 & \text{otherwise.} \end{cases}$$

The security of the execution mapping $X$ is defined such that $X$ is secure if for all states $s_1$ and $s_2$, equivalent input states yield equivalent output states (`is_safe` in the model):

$$\forall s_1, s_2 \in \mathcal{S} : s_1 \overset{\text{state}}{\equiv} s_2 \implies P_X(s_1) \overset{\text{state}}{\equiv} P_X(s_2). \quad (1)$$

That is, $X$ is secure, if after each step in a computation, the values of blinded registers and memory locations do not influence the blindedness of any component of the output state, nor the values of any unblinded value in the output, meaning that the attacker cannot infer anything about the blinded state.

*3) Load-store model:* The low-level system model described in Section VI-B2 is simple and easy to understand, but since the execution mapping $X$ has unmediated access to the state, there is no easy way to express the taint propagation rule from Section IV. In InstructionDecoder.fst we describe a higher-level model that allows us to better express statements about data flows between registers.

This model (the "load-store model") includes some microarchitectural details. Its execution mapping is defined in the function `decoding_execution_unit` in terms of two functions:

- *An instruction decoder*, a function that takes as input an instruction word, and returns a decoded instruction containing an opcode, a list of input registers, and a list of output registers–either normal registers or `pc`.
- *Instruction semantics*, a function that performs the actual computation, taking as input a decoded instruction and a list of `multiBlinded` register input values, and which returns a fault bit, a list of `multiBlinded` register output values with blindedness bits, and a list of memory operations, each of which indicates a load or store between a register and an address in memory.
- *A cache policy*, a function that accepts a set of cache line assignments and a memory operation, and returns a new set of cache line assignments.

The execution mapping then takes the instruction word, decodes it, reads the input operands from the initial system state, performs the computation, and if it does not raise a fault, increments `pc`, writes the results to registers, and performs the stores and loads, updating the cache line assignments. In the event of a fault, `pc` is set to a fixed value.

[2]To analyze the effects of features like speculation and variable-duration instructions, where instructions are not executed and committed sequentially, a more detailed microarchitecture-specific system model is needed where the model includes a wider range of internal processor state.

The decoded instructions never depend upon blinded data as, from Section VI-B2, attempts to execute a blinded value as an instruction results in a fault, and the execution mapping is never called. Therefore, the safety of the execution mapping can be demonstrated by analyzing only the instruction semantics, as shown in the load-store model's main safety theorem.

We define instruction semantics safety similarly to how we defined execution mapping safety in Equation (1): for all instruction words, executing the instruction semantics function with equivalent lists of input operand values raises a fault in both cases, or neither case raises a fault and both yield equivalent output operand lists and memory operation lists[3].

We then use this to prove the theorem `each_load_store_execution_unit_with_redacting_equivalent_instruction_semantics_is_safe` that an execution mapping defined by any instruction decoder and safe instruction semantics is safe, no matter what cache policy is in use.

We next use the model to show the safety of a concrete ISA.

*4) Model ISA:* Ideally, we would now apply the analysis above to a formal model extracted from the design of a real-world processor. However, this is a major undertaking in its own right that is beyond the scope of this paper. We therefore analyze a simplified model ISA with eight instructions in ISA.fst: `STORE`, `LOAD`, `BZ` (branch if zero), `ADD`, `SUB`, `MUL`, `AND`, and `XOR`.

Each instruction accepts two input registers and one output register—the exceptions being `STORE` and `LOAD`, which require only two registers, one for the memory address, and one for the source or destination register respectively, and `BZ`, whose output is always written to `pc`.

Each instruction specifies the blindedness of its outputs, as described in Section IV-D. By default, each instruction marks its output as blinded if any of its inputs are (consistently) blinded, or raises a fault where some of its inputs are blinded but have inconsistent blindedness tags. However, there are some special cases that deviate from this treatment in order to better capture the data dependencies of the instructions and handle modifications of visible state:

- `STORE` and `LOAD` instructions raise faults if their address input is blinded.
- `SUB` and `XOR` instructions yield an unblinded value zero if both their inputs are the same register.
- `MUL` and `AND` instructions yield an unblinded value zero if one of their inputs is an unblinded value zero.
- `BZ` raises a fault if its comparison input is blinded.

We then show in the model ISA's main safety theorem `sample_semantics_are_safe` that these instruction semantics, defined in `sample_semantics`, are safe as described in Section VI-B3, and that the architecture with these

[3]Memory operations are defined to be equivalent where they are between the same register and the same address in memory.

|  | Unmod. | BliMe-BOOM-1 |  | BliMe-BOOM-8 |  |
|---|---|---|---|---|---|
|  | Value | Value | Δ [%] | Value | Δ [%] |
| **Power** (W) | **37.783** | **38.137** | **+0.9** | **38.319** | **+1.4** |
| **Resources** |  |  |  |  |  |
| **LUTs** | **460480** | **478950** | **+4.0** | **501847** | **+9.0** |
| **Registers** | **357179** | **371024** | **+3.9** | **388956** | **+9.0** |

TABLE II. EFFECT OF BLIME-BOOM MODIFICATIONS ON POWER CONSUMPTION AND FPGA RESOURCE USAGE (UNMODIFIED BOOM VS. BLIME-BOOM-1 AND BLIME-BOOM-8).

semantics is therefore safe according to the definition given in Equation (1).

This demonstrates that the taint tracking approach proposed in Section IV-D is secure in general, and that the special cases that we have considered do not allow an attacker to violate the dataflow security definition from Equation (1). This means that, so long as the external peripherals with which an outside observer can interact do not expose blinded data, an observer cannot infer anything about the blinded data in the system, satisfying objective SR.

## VII. PERFORMANCE & RESOURCE USAGE EVALUATION

### A. *BliMe-BOOM*

Incorporating BliMe into BOOM affects logical complexity (hence maximum clock rate), and the number of cycles that certain operations will take due to the additional memory accesses required to fetch the blindedness tags. Therefore, we evaluate the overall performance of BliMe-BOOM using both the maximum clock rate, and the number of clock cycles taken to execute a program.

We synthesized BliMe-BOOM using the FireSim [26] FPGA-hosted simulation tool to measure the change in resource consumption and maximum clock rate. Table II shows the power estimate and FPGA resource usage provided by Vivado, indicating low overheads for both BliMe-BOOM-1 and BliMe-BOOM-8. Vivado timing analysis reported a Worst Negative Slack (WNS) of 0.018ns for both modified and unmodified cores, indicating no significant reduction to the maximum clock rate.

To determine the effect of BliMe-BOOM's memory modifications on performance, we ran the SPEC2017 Integer benchmark suite on both BliMe-BOOM variations using the `ref` workload. BliMe-BOOM-1 was run with 8GiB of main memory, and BliMe-BOOM-8 with 14GiB of main memory, so that both systems have 7GiB of addressable memory. This avoids the risk that the performance of the benchmarks is affected by differences in available memory. The results are shown in Figure 3; the average overhead across all benchmarks is **23%** for both BliMe-BOOM-1 and -8. This overhead, however, stems from the Chipyard limitation mentioned in Section V-B. As we show below in Section VII-B, appropriate memory optimization can substantially reduce this overhead.

### B. *Memory optimization*

In Section V-B, we mentioned that an implementation optimized for small blindedness tags can improve its performance by reducing the ratio of blindedness tag requests to data requests. We investigate this by simulating both the optimized and unoptimized designs using the gem5 simulator [38], where it is straightforward to reduce the memory request size for tags, and comparing their performance.

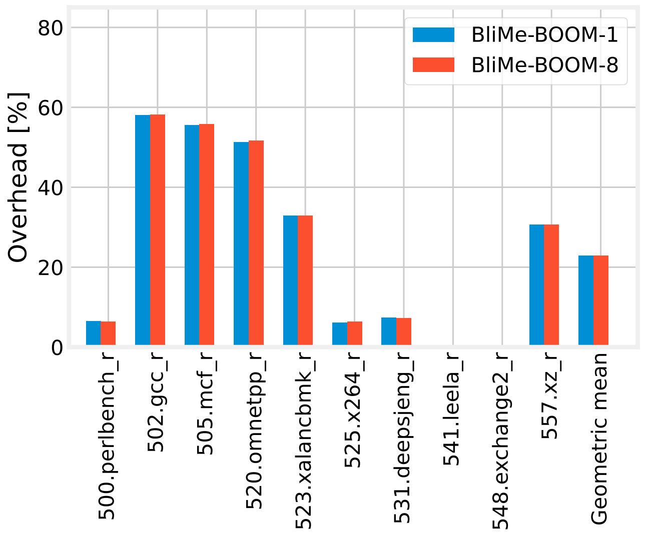

Fig. 3. Overhead of BliMe-BOOM-1 and BliMe-BOOM-8 relative to unmodified BOOM, as measured using SPEC2017. The average overhead was **23%** for both BliMe-BOOM-1 and BliMe-BOOM-8.

Gem5 is run in RISC-V full-system mode with the `O3` out-of-order processor model. The cache configuration was chosen to match that used in Section V-B: 16kB each for private L1D and L1I, and 256kB for private L2. Main memory was 7GB of Dual Channel DDR3-1600 DRAM. In addition, we modified the memory model to generate the extra requests used to read and write blindedness tags, with the ratio of blindedness tag to data request sizes being configurable as either 1:1 (as used in BliMe-BOOM) or 1:8 (as in the proposed optimization).

Since gem5 is much slower than native execution using a field-programmable gate array (FPGA), following the approach taken in prior work [68], we ran the SPEC CPU 2017 Integer benchmark suite with the `ref` workload in each configuration and measured the number of cycles taken to execute and commit 1 billion instructions after skipping the first 10 billion instructions in order to allow the benchmark proper to start. Table III compares the performance results for BliMe-BOOM-1, BliMe-BOOM-8 and BliMe-gem5. With the gem5 model configured in unoptimized mode, the average overhead of 25% is similar to that of both BliMe-BOOM-1 and -8 (23%). With the model configured in optimized mode, the overhead reported by gem5 is reduced to **8%**. The detailed benchmark results comparing the two configurations is shown in Figure 4. Therefore, we conclude that if the optimizations are implemented in hardware (by making the needed modifications to the memory subsystem in the Chipyard SoC as mentioned in Section V-B), similar overhead reductions can be achieved. As a result, for a tag-to-data size ratio of 1:8, 8% is a fairer representation of BliMe's overhead in real deployments by hardware manufacturers. Use of tag caches [25] would reduce overheads even further. The earlier figure of 23%, on the other hand, corresponds to a tag-to-data size ratio of 1:1.

The closest comparisons to BliMe, oblivious instruction

| Implementation | Avg. overhead (%) |
|---:|---:|
| BliMe-BOOM-1 | **23** |
| BliMe-BOOM-8 | **23** |
| BliMe-gem5 | **25** |
| BliMe-gem5 Optimized | **8** |

TABLE III. AVERAGE OVERHEADS OF RUNNING SPEC2017 ON BLIME-BOOM-1 AND -8, BLIME-GEM5 AND BLIME-GEM5 OPTIMIZED. THE AVERAGES ARE CALCULATED FOR THE FULL BENCHMARKS ON BLIME-BOOM-1 AND -8, AND FOR 1 BILLION INSTRUCTIONS ON BLIME-GEM5 AND BLIME-GEM5 OPTIMIZED. THE GEM5 SIMULATION OF THE UNOPTIMIZED SYSTEM SHOWS AN AVERAGE OVERHEAD OF 25%, SIMILAR TO THAT OF BOTH BLIME-BOOM-1 AND -8 (23%).

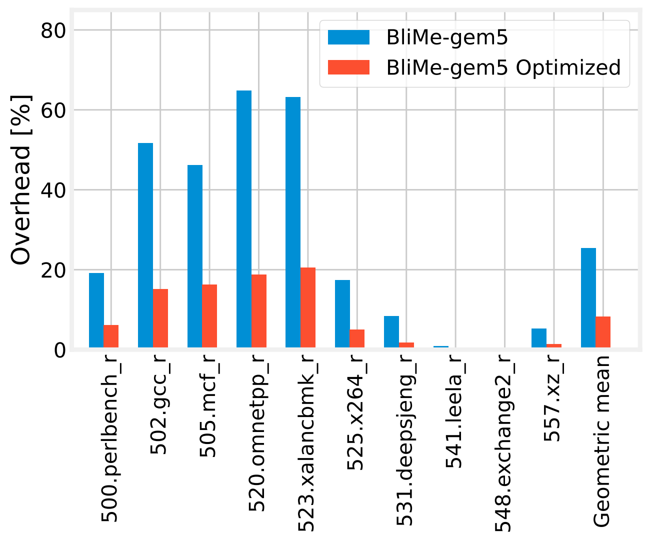

Fig. 4. Overhead before and after applying the optimization from Section V-B, as measured using gem5 simulator to execute 1 billion instructions of SPEC2017. The optimization reduces the average overhead on gem5 from 25% to 8%.

set architecture (OISA) [67] and FHE, suffer from significantly higher overheads. OISA's fastest benchmarks (matrix-mult and neural-net) incur approximately 35% overhead with larger-than-cache inputs, whereas FHE is several orders of magnitude slower [62]. Overall, given the moderate impacts on performance and resource usage discussed in Sections VII-A and VII-B, we conclude that requirement PR is satisfied.

## VIII. COMPATIBILITY EVALUATION

For backwards compatibility with existing code (requirement CR), two types of code are important: code that processes exclusively non-blinded data, and code that is already side-channel resistant and processes blinded data.

BliMe's taint tracking policy (Section IV-D) changes the behavior of the processor only where an instruction depends upon a blinded input value. Therefore, code that exclusively processes non-blinded data is unaffected.

Code that processes blinded data will fault if it attempts to modify an observable output in a way that the processor determines to be dependent on the blinded data. To evaluate whether the BliMe implementations have met requirement CR, we must therefore determine whether side-channel-resistant code will comply with the BliMe taint-tracking policy.

BliMe faults when blinded data attempts to flow to non-blindable observable outputs (Section IV-D). Existing side-channel-resistant code already prevents the program counter and the addresses of memory operations from being affected by blinded data, e.g., as in [7]. Data flows of blinded data to other visible state, execution time and fault signals (Section IV-D) are prevented by design. Therefore, side-channel-resistant code is compatible with BliMe's taint-tracking policy, so long as only sensitive data is blinded. We successfully ran stream cipher encryption/decryption with a blinded key using the TweetNaCl [8] library, demonstrating the backwards compatibility of BliMe.

However, the CPU cannot identify cases where the result of a computation remains blinded even though it no longer contains any sensitive data. For example, it will not be possible to branch on the result of the decryption/verification of an authenticated encryption, meaning that its cryptographic application programming interfaces (APIs) will need to be modified so that its control flow does not depend on whether the verification was successful. In practice, this means that any computation must always continue as though verification was successful, with any failure being indicated by a flag that is returned to the client (Section IX-D).

## IX. FUTURE WORK

### A. Compiler support to improve deployability

The security guarantees of BliMe are enforced solely by the hardware modifications without requiring any compiler support. We saw that some existing side-channel resistant code successfully runs on BliMe (Section VIII). Developers can write new BliMe-compliant code. However, this can be a challenge, particularly for larger programs because 1) manual data-flow tracking can be too complex, and 2) BliMe-unaware compilers might inadvertently induce data-flow dependencies not evident in the source language. Consequently, compiler support can increase deployability by (a) *verifying* code compatibility, and where possible (b) automatically *transforming* non-compliant code to an equivalent compliant variant, or (c) semi-automatically transforming the code by identifying problematic pieces of code to be manually rewritten by the developer.

To verify compliance, one approach is to use binary analysis. This has a number of drawbacks. Binary lifting is itself error-prone [28], and could induce uncertainty. Focusing on binary representation makes it difficult to generate useful error messages. Instead, we opt for augmenting the compiler so that we can use the intermediate representation (IR) for transformations, provide useful error messages through the front-end, and control the machine-code generation.

To transform code, we can employ known rules for safe coding like *cryptocoding* [5], and use compiler-specific approaches that aid both analysis and verification. Specifically, we can use techniques such as *function cloning* to improve the accuracy of the static data-flow tracking and to avoid instrumentation of data-flows that at run-time are not necessarily blinded. For instance, a function could be called in multiple places, both such that its arguments are never tainted, and that they may be tainted. By cloning the function we can avoid needlessly instrumenting and tainting the non-blinded data-flow and instead only taint the cloned variant of the function. We are currently exploring both the adaption of existing software-only approaches such as Constantine [9], and implementing BliMe-specific analysis and transformations.

Because complete program analysis is undecidable [47], we cannot guarantee that any approach detects all compliant code. Given memory-safe code, we require a sound analysis that rejects all non-compliant code and accepts only code that will not result in a taint-tracking policy violation fault at run-time.

## B. Handling large numbers of clients

BliMe as described above can handle $2^n - 1$ simultaneous clients with an $n$-bit blindedness tag. This means that the ability to handle larger numbers of clients will require a logarithmic increase in memory overhead. This can be made configurable at boot-time, allowing servers to trade main memory capacity for a larger number of simultaneous clients. Another approach, however, is to overcome this overhead by multiplexing the same blindedness tag value across multiple clients, and using an OS trusted by clients—e.g., by using remote attestation of a well-known certified OS—to prevent data from flowing between programs that share blindedness tag values. In the extreme case, this reduces to a single-bit blindedness tag, as implemented in BliMe-BOOM-1.

This OS has several tasks. On context switch, the OS asks the encryption engine to export the current client key by "sealing" it (i.e., encrypting it using a key known only to the encryption engine), and provides a previously sealed key, corresponding to the incoming process, which the encryption engine can unseal and set as the new current client key. Thus, even though the OS never sees any client key in the clear, it must be trusted to load the correct sealed key onto the encryption engine.

The normal process isolation features of the OS ensure that a client cannot access another client's blinded data directly or any of the client session keys held by the OS. Because different processes may share blindedness tag values, the OS must further ensure that an application cannot transfer blinded data to other applications via system calls or other inter-process communication (IPC) mechanisms. Preventing a client from accessing another client's blinded data ensures two things: 1) an application serving one client cannot use the encryption engine to encrypt and unblind blinded data originating from another client, and 2) computation can never use blinded data belonging to two different clients, since there would be no clear way to determine the ownership of the result. Therefore, processed data is only encrypted and exported with a key corresponding to the client from which the input data was originally imported. Note that we only rely on the OS to prevent clients from having *direct* access to other clients' blinded data. We do not rely on the OS code being side-channel resistant, because blinded data is protected from side channels by the hardware.

## C. Enabling safe local processing

BliMe extensions can be usefully applied on the local machine as well. Since an application processing blinded data cannot infer anything about the data other than its length, it can safely process data belonging to other users or applications.

For example, the OS can allow an application to read data from a file that is normally inaccessible to the application with the constraint that any data read will be marked as blinded. This makes it possible to build useful computational pipelines while strongly adhering to the principle of least privilege.

## D. Handling secret-dependent faults

In BliMe, a fault's occurrence cannot depend on a secret value, since the fact that it occurs (or not) can leak information. For example, if a div-by-zero fault will occur if the divisor is a blinded data item with a value zero, and this faults leads to an interrupt handler being called, the change in control flow will reveal that the blinded value was zero. On the other hand, if it does not occur, it reveals that the value was not zero. We therefore do not allow blinded values to be used in such situations. To avoid this limitation, we might instead suppress faults depending on blinded data, so that the control flow remains *as if the fault did not occur*. The client can be informed of this fault and that the computation results are invalid by setting a blinded bit in some protected storage (e.g., a special register) when the fault occurs, so that the software can convey this bit to the client as part of the returned encrypted results.

## E. Defending against other attacks

Rowhammer [30] is a vulnerability in modern dynamic random-access memory (DRAM) modules that threatens the integrity of data. Due to the high proximity of DRAM cells, toggling a row of cells at a sufficiently high rate can result in bit flips in adjacent rows. Exploits of this phenomenon by a remote adversary have been continuously demonstrated despite the number of proposed defenses [41]. We make the common and reasonable assumption that reading data from any address in memory retrieves the last value that was written to that address; this includes the extra blindedness tag. Rowhammer is an orthogonal vulnerability requiring orthogonal defenses to ensure memory integrity. Other fault-injection attacks based on dynamic voltage and frequency scaling (DVFS), such as CLKscrew [57], V0LTpwn [27] and Plundervolt [40], produce similar attack patterns and are also out of scope.

Attacks that require physical access to the server are also out of scope. Physical access enables full read-write side-band access to memory through direct connection to the DRAM bus. It also facilitates more powerful side-channel attacks, such as those relying on power [32], electromagnetic [16], [37] or temperature measurements [24], or fault-injection attacks, such as VoltPillager [13], that can break HSM confidentiality and integrity guarantees. We leave an adversary model that includes physical access to future work.

## X. Related work

**Taint tracking.** A large body of work exists on taint tracking, also called dynamic information flow tracking (DIFT) [55]. Hu *et al.* [23] present a survey that includes several hardware-based taint-tracking techniques with varying goals and security/performance trade-offs, and at different abstraction layers. Speculative Taint Tracking [68] applies taint tracking to the results of speculatively executed instructions to prevent them leaking information. Tiwari *et al.* [58] propose taint tracking at the gate level, and use it to create a processor that is able to track all information flows, but has a limited ISA and suffers from large overheads. Taint tracking can also be performed purely in software. Data flow integrity is a form of taint tracking that protects software against non-control data attacks by using reaching definitions analysis [11]. Pointer tainting [65] is another defense against non-control data attacks that taints user input and detects an attack when a tainted value is dereferenced as a pointer.

**Data-oblivious execution.** Preventing side-channel leakage requires covering several observable outputs. Several algorithms have been proposed to obfuscate data-dependent memory access patterns [17], [54], [59], [4]. However, they all come at a significant cost to performance. Other work has focused on making code constant-time to prevent leakage through execution timing. An example is Constantine [9], which extends LLVM to compile code into constant-time binaries. It relies, however, on dynamic analysis to identify vulnerable code for transformation. This can be imprecise as full execution path coverage is not guaranteed, potentially leading to some vulnerable code not being transformed.

Lee *et al.* [33] propose DOVE to protect sensitive data used in outsourced computation from side channels. DOVE uses a frontend to transform the client application code to a custom data-oblivious representation called a Data Oblivious Transcript (DOT). The DOT is then sent to a trusted interpreter (the backend) on the server, which verifies that the DOT is data oblivious and then runs it on the sensitive data. The trusted interpreter must be run within a TEE, such as an Intel SGX enclave, as it is part of the trusted code base (TCB).

Yu *et al.* [67] develop an OISA that performs run-time taint tracking of sensitive values and adds a duplicate set of instructions to the ISA. Each operand of the additional instructions is defined as either safe or unsafe. Using any tainted values as unsafe operands results in a fault. The hardware guarantees that computation is oblivious to safe operands and, therefore, that any sensitive values used as safe operands are not leaked through side channels. OISA offers taint and untaint instructions and relies on the application code to use them correctly to taint/untaint sensitive values during computation. Consequently OISA is *not applicable for our usage scenario* described in Section III-A, where outsourced computation is carried out by potentially *untrusted, third-party*, application code, for multiple simultaneous clients. Although OISA could use remote attestation to verify the server-side application code, remote attestation of arbitrary third-party application code is neither realistic nor scalable. Furthermore, software vulnerabilities in the application code can allow adversaries to untaint arbitrary sensitive values through the OISA's untainting instruction, which is exposed to any software running on the main CPU.

**Point solutions for side-channel attacks.** The literature contains a variety of side-channel attacks that leak information through the processor's caches [66], [21], [42], [22], [20]. Defenses against these attacks rely on temporal or spatial isolation between processes; the cache is either flushed on context switches or is partitioned in such a way that each process uses a separate fixed portion of the cache [44], [63]. However, this results in unnecessary overhead when processing non-sensitive data. Other methods change the cache architecture or replacement policy but also suffer from unnecessary overheads [51], [64]. The cache attacks mentioned above are usually used as building blocks to create covert channels for more sophisticated attacks such as Meltdown and Spectre [36], [31]. In response, a range of defenses have been proposed to stop these sophisticated attacks [68], [19]. However, they do not address the main source of information leakage (which is the data-dependent memory access pattern) but rather provide point solutions for specific attacks.

## XI. Conclusion

We introduced BliMe, a new approach to outsourced computation that uses hardware extensions to ensure that clients' sensitive data is not leaked from the system, even if an attacker is able to run malware, exploit software vulnerabilities, or use side channel attacks. BliMe does this while maintaining compatibility with existing side-channel-resistant code, and without reducing performance significantly. In designing BliMe, we follow the design pattern of using a separate, discrete hardware security component in conjunction with the main CPU, common on both servers [60] and end user devices [39]. By using such a remotely attestable fixed-function HSM in combination with taint-tracking ISA extensions, BliMe can provide functionality similar to that of fully homomorphic encryption, but achieving native-level performance by replacing cryptography with hardware enforcement.

## Acknowledgements

This work is supported in part by Natural Sciences and Engineering Research Council of Canada (grant number RGPIN-2020-04744), Intel (in the context of the Private AI consortium), the Government of Ontario, and the Academy of Finland (decision 339514). Views expressed in the paper are those of the authors and do not necessarily reflect the position of the funders. Our performance evaluation was facilitated in part by a gift of cloud credits from Amazon Web Services.

## References

[1] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[2] Apple, "Secure enclave," https://support.apple.com/en-ca/guide/security/sec59b0b31ff/web, Apple Support, 2022.

[3] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[4] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi, "OptORAMa: Optimal oblivious RAM," in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*, Berlin, Heidelberg, 2020, pp. 403–432.

[5] J.-P. Aumasson, "Cryptocoding," https://github.com/veorq/cryptocoding, Github, 2022.

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *Proceedings of the Design Automation Conference*, New York, NY, USA, 2012, pp. 1216–1225.

[7] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, “High-speed high-security signatures,” *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012.

[8] D. J. Bernstein, B. van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers, “TweetNaCl: A crypto library in 100 tweets,” in *Progress in Cryptology - LATINCRYPT 2014*, D. F. Aranha and A. Menezes, Eds., Cham, 2015, vol. 8895, pp. 64–83.

[9] P. Borrello, D. C. D’Elia, L. Querzoni, and C. Giuffrida, “Constantine: Automatic side-channel resistance using efficient control and data flow linearization,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2021, pp. 715–733.

[10] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, “Software grand exposure: SGX cache attacks are practical,” in *Proceedings of the USENIX Conference on Offensive Technologies*, USA, 2017.

[11] M. Castro, M. Costa, and T. Harris, “Securing software by enforcing data-flow integrity,” in *Proceedings of the Symposium on Operating Systems Design and Implementation*, USA, 2006, pp. 147–160.

[12] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, “SgxPectre: Stealing intel secrets from SGX enclaves via speculative execution,” in *Proceedings of the IEEE European Symposium on Security and Privacy*, 2019, pp. 142–157.

[13] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. Oswald, and F. D. Garcia, “VoltPillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface,” in *Proceedings of the USENIX Security Symposium*, 2021, pp. 699–716.

[14] V. Costan and S. Devadas, “Intel SGX explained,” Cryptology ePrint Archive, Report 2016/086, 2016.

[15] V. Costan, I. Lebedev, and S. Devadas, “Sanctum: Minimal hardware extensions for strong software isolation,” in *Proceedings of the USENIX Security Symposium*, USA, 2016, pp. 857–874.

[16] K. Gandolfi, C. Mourtel, and F. Olivier, “Electromagnetic analysis: Concrete results,” in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*, Berlin, Heidelberg, 2001, pp. 251–261.

[17] O. Goldreich and R. Ostrovsky, “Software protection and simulation on oblivious RAMs,” *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[18] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, “Cache attacks on intel SGX,” in *Proceedings of the European Workshop on Systems Security*, New York, NY, USA, 2017, pp. 1–6.

[19] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, “KASLR is dead: Long live KASLR,” in *Proceedings of the International Symposium on Engineering Secure Software and Systems*, Cham, 2017, pp. 161–176.

[20] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, “Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2016, pp. 368–379.

[21] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, “Flush+flush: A fast and stealthy cache attack,” in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Berlin, Heidelberg, 2016, pp. 279–299.

[22] D. Gruss, R. Spreitzer, and S. Mangard, “Cache template attacks: Automating attacks on inclusive last-level caches,” in *Proceedings of the USENIX Security Symposium*, USA, 2015, pp. 897–912.

[23] W. Hu, A. Ardeshiricham, and R. Kastner, “Hardware information flow tracking,” *ACM computing surveys*, vol. 54, no. 4, pp. 1–39, 2021.

[24] M. Hutter and J.-M. Schmidt, “The temperature side channel and heating fault attacks,” in *International Conference on Smart Card Research and Advanced Applications*, 2014.

[25] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son, and A. T. Markettos, “Efficient Tagged Memory,” in *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, 2017, pp. 641–648.

[26] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic, “FireSim: FPGA-Accelerated cycle-exact scale-out system simulation in the public cloud,” in *Proceedings of the International Symposium on Computer Architecture*, 2014, pp. 29–42.

[27] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, “V0LTpwn: Attacking x86 processor integrity from software,” in *Proceedings of the USENIX Security Symposium*, 2020, pp. 1445–1461.

[28] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, “Testing intermediate representations for binary analysis,” in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, Urbana-Champaign, IL, USA, 2017, pp. 353–364.

[29] T. Kim and Y. Shin, “ThermalBleed: A practical thermal side-channel attack,” *IEEE Access*, vol. 10, pp. 25 718–25 731, 2022.

[30] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors,” in *Proceedings of the International Symposium on Computer Architecture*, Minneapolis, Minnesota, USA, 2014, pp. 361–372.

[31] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2019, pp. 1–19.

[32] P. C. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” in *Proceedings of the International Cryptology Conference on Advances in Cryptology*, Berlin, Heidelberg, 1999, pp. 388–397.

[33] H. B. Lee, T. M. Jois, C. W. Fletcher, and C. A. Gunter, “DOVE: A data-oblivious virtual environment,” in *Proceedings of the Network and Distributed System Security Symposium*, Virtual, 2021.

[34] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, “Inferring fine-grained control flow inside SGX enclaves with branch shadowing,” in *Proceedings of the USENIX Security Symposium*, USA, 2017, pp. 557–574.

[35] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, “PLATYPUS: Software-based power side-channel attacks on x86,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2021, pp. 355–371.

[36] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, and R. Strackx, “Meltdown: Reading kernel memory from user space,” *Communications of the ACM*, vol. 63, no. 6, pp. 46–56, 2020.

[37] J. Longo, E. De Mulder, D. Page, and M. Tunstall, “SoC it to EM: ElectroMagnetic side-channel attacks on a complex system-on-chip,” in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*, Berlin, Heidelberg, 2015, pp. 620–640.

[38] J. Lowe-Power *et al.*, “The gem5 simulator: Version 20.0+,” *CoRR*, vol. abs/2007.03152, 2020. [Online]. Available: https://arxiv.org/abs/2007.03152

[39] D. Melotti, M. Rossi-Bellom, and A. Continella, “Reversing and fuzzing the google titan m chip,” in *Reversing and Offensive-Oriented Trends Symposium*, New York, NY, USA, 2021, pp. 1–10.

[40] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, “Plundervolt: Software-based fault injection attacks against intel SGX,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2020.

[41] O. Mutlu and J. S. Kim, “RowHammer: A retrospective,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2020.

[42] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: The case of AES,” in *Proceedings of the Cryptographers’ Track at the RSA Conference on Topics in Cryptology*, Berlin, Heidelberg, 2006, pp. 1–20.

[43] R. Paccagnella, L. Luo, and C. W. Fletcher, “Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical,” in *Proceedings of the USENIX Security Symposium*, 2021, pp. 645–662.

[44] D. Page, “Partitioned cache architecture as a side-channel defence mechanism,” Cryptology ePrint Archive, Report 2005/280, 2005.

[45] S. Pinto and N. Santos, “Demystifying Arm TrustZone: A comprehensive survey,” *ACM Computing Surveys*, vol. 51, no. 6, pp. 1–36, 2019.

[46] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, “Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks,” in *Proceedings of the USENIX Security Symposium*, 2021, pp. 1451–1468.

[47] H. G. Rice, “Classes of recursively enumerable sets and their decision problems,” *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[48] RISC-V International, “Spike RISC-V ISA simulator,” https://github.com/riscv-software-src/riscv-isa-sim, RISC-V Software, 2022.

[49] R. Rojas, “How to make zuse’s z3 a universal computer,” *IEEE Annals of the History of Computing*, vol. 20, no. 3, pp. 51—54, 1998.

[50] K. Ryan, “Hardware-backed heist: Extracting ECDSA keys from qualcomm’s TrustZone,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2019, pp. 181–194.

[51] D. Sanchez and C. Kozyrakis, “The ZCache: Decoupling ways and associativity,” in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, USA, 2010, pp. 187–198.

[52] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, “ZombieLoad: Cross-privilege-boundary data sampling,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2019, pp. 753–768.

[53] B. A. Shivakumar, J. Barnes, G. Barthe, S. Cauligi, C. Chuengsatiansup, D. Genkin, S. O’Connell, P. Schwabe, R. Q. Sim, and Y. Yarom, “Spectre declassified: Reading from the right place at the wrong time,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2023, pp. 185–202.

[54] E. Stefanov, M. V. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, “Path ORAM: An extremely simple oblivious RAM protocol,” *Journal of the ACM*, vol. 65, no. 4, pp. 1–26, 2018.

[55] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, “Secure program execution via dynamic information flow tracking,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2004, pp. 85–96.

[56] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, “Dependent types and multi-monadic effects in F*,” in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, 2016, pp. 256–270.

[57] A. Tang, S. Sethumadhavan, and S. Stolfo, “CLKSCREW: Exposing the perils of security-oblivious energy management,” in *Proceedings of the USENIX Security Symposium*, USA, 2017, pp. 1057–1074.

[58] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, “Complete information flow tracking from the gates up,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2009, pp. 109–120.

[59] S. Tople, H. Dang, P. Saxena, and E.-C. Chang, “PermuteRam: Optimizing oblivious computation for efficiency,” Cryptology ePrint Archive, Report 2017/885, 2017.

[60] Trusted Computing Group, “TPM 2.0 library,” https://trustedcomputinggroup.org/resource/tpm-library-specification/, 2019.

[61] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, “Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution,” in *Proceedings of the USENIX Security Symposium*, 2018, pp. 991–1008.

[62] A. Viand, P. Jattke, and A. Hithnawi, “SoK: Fully homomorphic encryption compilers,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2021, pp. 1092–1108.

[63] Z. Wang and R. B. Lee, “New cache designs for thwarting software cache-based side channel attacks,” in *Proceedings of the International Symposium on Computer Architecture*, New York, NY, USA, 2007, pp. 494–505.

[64] ——, “A novel cache architecture with enhanced performance and security,” in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, USA, 2008, pp. 83–93.

[65] J. Xu and N. Nakka, “Defeating memory corruption attacks via pointer taintedness detection,” in *Proceedings of the International Conference on Dependable Systems and Networks*, USA, 2005, pp. 378–387.

[66] Y. Yarom and K. Falkner, “FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack,” in *Proceedings of the USENIX Security Symposium*, USA, 2014, pp. 719–732.

[67] J. Yu, L. Hsiung, M. El’Hajj, and C. W. Fletcher, “Data oblivious ISA extensions for side channel-resistant and high performance computing,” in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, 2019.

[68] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, “Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data,” in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2019, pp. 954–968.

[69] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, “TruSense: Information leakage from TrustZone,” in *Proceedings of the IEEE Conference on Computer Communications*, 2018, pp. 1097–1105.

[70] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, “SonicBOOM: The 3rd generation berkeley out-of-order machine,” *Proceedings of the Workshop on Computer Architecture Research with RISC-V*, 2020. [Online]. Available: https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf

## Appendix A
## Examples of BliMe-compliant applications

In this section we present two examples of code that adheres to constant-time principles and can therefore run without faulting on BliMe. We also show how to use BliMe’s blinding and unblinding operations with these examples. Both examples were adapted from the OISA work by Yu *et al.* [67]. The first example is data-oblivious matrix multiplication and requires no changes. The second is a function to find the maximum value in an array. We present the initial code and then show how to manually transform it to make it data-oblivious.

### A. *Matrix multiplication*

Listing 1 shows the code for matrix multiplication. The code is already data-oblivious, i.e., adheres to the restrictions in Section IV-E. It therefore does not require any changes. In Listing 1, we also show how encrypted data from the client is preprocessed to decrypt-and-blind it, and how the result is postprocessed to encrypt-and-unblind it.

### B. *Finding the maximum*

Listing 2 shows non-data-oblivious code for finding the maximum value in an array of integers. The blinding and unblinding operations are similar to those in Listing 1, and we therefore skip them for brevity.

Initially, the only blinded data is that inside `arr`. In the first iteration, blinded data from `arr` gets loaded into `max_val` on line 6. In the second iteration, the blinded value in `max_val` is used in a branching condition on line 4, resulting in a BliMe violation and a fault.

We can make this code data-oblivious by linearizing the if condition using predicated execution. The transformed code is shown in Listing 3.

```

// int *A, *B, *C;
// with sizes szA, szB, szC, respectively

// Here, the input matrices A and B are
// populated with unblinded encrypted data
// from the client, e.g., through some
// network interface

// We first decrypt-and-blind A and B:
blind(A, szA);
blind(B, szB);

// Now, A and B contain blinded plaintext

int C_nrow = A_nrow;
int C_ncol = B_ncol;
assert (A_ncol == B_nrow);
for(int i = 0; i < C_nrow; i++){
  for(int j = 0; j < C_ncol; j++){
    for(int k = 0; k < A_ncol; k++){
      C[i*C_ncol + j] +=
          A[i * A_ncol + k] * B[k * B_ncol + j];
    }
  }
}

// Now, C contains the blinded plaintext result
// We must then unblind C ...

unblind(C, szC);

// ... which gives us the result as
// unblinded ciphertext that we can then
// send back to the client

```

Listing 1. Data-oblivious code for matrix multiplication.

```
int FindMax(int arr[] /*blinded*/,
            int N     /*unblinded*/){
  int max_val = -1;
  for (int i = 0; i < N; i++){
    if (arr[i] > max_val){
      max_val = arr[i];
    }
  }
}
```

Listing 2. Non-data-oblivious code for finding the maximum value in an array of integers.

## Appendix B
## Artifact Appendix

### A. Description & Requirements

BliMe's evaluation consists of three parts: security (Section VI), performance (Section VII - Table III, Figures 4) and resource usage (Section VII - Table II).

```
int FindMax(int arr[] /*blinded*/,
            int N     /*unblinded*/){
  int max_val = -1;
  for (int i = 0; i < N; i++){
    int predicate = arr[i] > max_val;
    max_val = (predicate * arr[i]) |
              (!predicate * max_val);
  }
}
```

Listing 3. Data-oblivious code for finding the maximum value in an array of integers after manual transformation.

- **Security**: The security evaluation is done by running the provided model in F* and confirming that it passes verification.
- **Cycle-accurate performance**: The performance evaluation is done by running SPEC CPU17 on two gem5 implementations, BliMe-gem5 and -gem5 Optimized. These can be run entirely on commodity hardware but are only used to run 11 billion instructions of each SPEC CPU17 benchmark; running the benchmarks to completion requires an infeasible amount of time. The performance of BliMe-gem5 is an estimate of the performance of BliMe-BOOM-1 and -8. This is because the only performance overhead introduced by BliMe is the additional memory accesses to tags, which take the same number of cycles for both BliMe-BOOM-1 and BliMe-BOOM-8. BliMe-gem5 Optimized corresponds to an optimized version that is not possible on BliMe-BOOM due to a limitation in Chipyard not BliMe (Section V.B.2).
- **Resource usage & clock frequency**: The power, resource usage and clock frequency overheads are obtained by synthesizing BliMe-BOOM-1 and -8 using Xilinx Vivado on an Amazon Web Services (AWS) instance. This is done by running FireSim scripts.

*1) How to access:* Our artifact is available at https://github.com/ssg-research/BliMe, and archived at https://doi.org/10.5281/zenodo.10161487.

*2) Hardware dependencies:* Obtaining the overheads requires the use of AWS. This is because we use FireSim scripts that must be run on AWS.

*3) Software dependencies:* F* and gem5 implementations were tested on Ubuntu 20.04.

*4) Benchmarks:* We use the SPEC CPU 2017 benchmark suite for the gem5 performance evaluation, which must be obtained separately from https://www.spec.org/cpu2017/.

### B. Artifact Installation & Configuration

Docker is required for security evaluation using F*. Installation then proceeds according to the README.

The full FireSim evaluation requires an AWS instance running the AWS FPGA Developer AMI. The README in the repository describes how to start and configure a suitable instance. Installation then proceeds according to the README.

The gem5 evaluation requires a Ubuntu Linux system. Installation then proceeds according to the README in the repository.

## C. Major Claims

- (C1): A simplified model ISA with BliMe extensions is formally verified to maintain the confidentiality of blinded data.
- (C2): An unoptimized BliMe implementation incurs a 25% performance overhead. Optimization reduces this to 8%.
- (C3): BliMe has low power and FPGA resource usage overheads.

## D. Evaluation

*1) Experiment (E1):* [1 human-minute + 1–2 compute-hours]: This machine-checks the F* model to verify that the confidentiality of blinded data cannot be broken (C1).

*[Preparation]* Build the Docker container from the `model/` directory of the repository.

*[Execution]* Run the Docker container built in *[Preparation]* without arguments.

*[Results]* The container will output the message `All verification conditions discharged successfully`.

*2) Experiment (E2):* [30 human-minutes + 24 compute-hours]: This runs 11 billion instructions of each SPEC CPU 2017 benchmark on BliMe-gem5 and BliMe-gem5 Optimized. The first 10 billion instructions are used to warm-up the system. The results of E2 provide an estimate for claim C2 in terms of number of cycles. We show in E3 that the time for each cycle (clock frequency) does not change, which means that the performance overhead is dependent *only* on the change in the *number* of cycles, i.e., E2.

*[Preparation]* Follow the instructions in the `gem5` folder's README file.

*[Execution]* Follow the instructions in the README file. This will run the 10 SPEC CPU 2017 Integer benchmarks on the unmodified gem5, BliMe-gem5 and BliMe-gem5 Optimized. Two `tmux` sessions will be spawned per benchmark: one (with `_run_` in its name) to run the gem5 simulation and one (with `_telnet_`) to `telnet` into the gem5 simulation and run the benchmark. Once the benchmark is complete, the gem5 simulation will exit. This can be checked by attaching to the `_run_` tmux session using `tmux a -t <name-of-tmux-session>`.

**Note:** Benchmarks can rarely crash prematurely. We have experienced this on both modified and unmodified gem5. Rerunning the benchmark should fix the issue. Please refer to the README-single file for information on how to rerun a single benchmark.

*[Results]* Once all the gem5 simulation have exited, the results will be in the `m5out/stats.txt` files in each benchmark directory. Follow the instructions at the end of the README file to extract the IPC numbers.

*3) Experiment (E3):* [30 human-minutes + 18 compute-hours]: This synthesizes BliMe-BOOM-1 and -8 using FireSim on AWS. The synthesis will produce power and resource usage reports that support claim C3. It will also produce a timing report that shows no reduction in maximum clock frequency, supporting our argument in Section B-D2.

*[Preparation]* Follow the instructions in the `firesim` folder's README file to set up the AWS EC2 instance.

*[Execution]* Follow the instructions in the README file. This will clone our modified FireSim repository and run the scripts to build the FPGA bitstream. NB: Building the bitstream will use several hundred US Dollars' worth of computation. Following the README's additional instructions to run the full SPEC 2017 benchmark will cost approximately US$5000.

*[Results]* The power, timing and resource usage reports will be available in `firesim/deploy/results-build/<timestamp-config>/<design>/build/reports/`. Power & resource usage results can be found in `<timestamp>.SH_CL_final_power.rpt` under "Total On-Chip Power" & "On-Chip Components". Clock frequency results can be found in `<timestamp>.SH_CL_final_timing_summary.rpt` under "WNS(ns)".

## E. Known Issues

- Attempting to recursively clone submodules for the FireSim experiments can result in an error about unreachable submodules. Please do not attempt to handle submodules directly. The build scripts used in the README instructions will fetch the submodules correctly.