Whole-Program Control-Flow Path Attestation

Nikita Yadav
Indian Institute of Science
Bangalore, India
nikitayadav@iisc.ac.in

Vinod Ganapathy
Indian Institute of Science
Bangalore, India
vg@iisc.ac.in

ABSTRACT

Path attestation is an approach to remotely attest the execution of a program $\mathcal{P}$. In path attestation, a prover platform, which executes $\mathcal{P}$, convinces a remote verifier $\mathcal{V}$ of the integrity of $\mathcal{P}$ by recording the path that $\mathcal{P}$ takes as it executes a particular input. While a number of prior techniques have been developed for path attestation, they have generally been applied to record paths only for parts of $\mathcal{P}$'s execution. In this paper, we consider the problem of *whole program control-flow path attestation*, *i.e.,* to attest the execution of the entire program path in $\mathcal{P}$. We show that prior approaches for path attestation use sub-optimal techniques that fundamentally fail to scale to whole program paths, and impose a large runtime overhead on the execution of $\mathcal{P}$. We then develop Blast, an approach that reduces these overheads using a number of novel approaches inspired by prior work from the program profiling literature. Our experiments show that Blast makes path attestation more practical for use on a wide variety of embedded programs.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**.

KEYWORDS

Control-flow attestation; trusted computing; ARM TrustZone

ACM Reference Format:
Nikita Yadav and Vinod Ganapathy. 2023. Whole-Program Control-Flow Path Attestation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3616687

1 INTRODUCTION

Attestation allows a remote *verifier* to establish trust in the integrity of a system, called the *prover*. An agent on the prover *measures* the state of the system to be verified, and sends these measurements to the verifier, who uses them to reason about the integrity of the prover. Because the verifier does not trust the prover, the measurements are generally tied to a root of trust on the prover platform, such as trusted platform modules (TPM) [43] on early systems, or other hardware-based trusted execution environments (TEE) such as the ARM TrustZone [6] or Intel SGX [31] in more recent systems. Integrity measurements can attest to a variety of properties, such as whether the prover booted up with a particular software stack [5, 25], that the system satisfies certain properties [16, 17, 32, 42], or executed certain operations [47].

Our focus in this paper is on *control-flow path attestation*, or simply, *path attestation*. In path attestation, the verifier supplies an input to the prover (which is a program). The goal of the prover is to produce a measurement that allows the verifier to precisely determine the control-flow path taken by the prover program as it processes that input. The main advantage of path attestation is that it provides the verifier with a non-repudiable proof of execution of the prover on that input. This form of attestation is particularly valuable for embedded systems, in which a remote verifier can obtain a proof that some requested action was indeed performed by an embedded device. For example, path attestation can be used to assure a verifier that a syringe pump embedded within a patient has indeed delivered the desired dosage of a drug [2], or that a robot arm has indeed completed a task in response to a command from a remote operator [47]. Because path attestation works on *individual runs* of the program, it attests the integrity of each run, and makes any attacks on the program more readily detectable by the verifier. Path attestation can also be seen as a lighter-weight alternative to verifiable computation [50].

Prior work on path attestation (*e.g.,* [2, 3, 21, 29, 39, 47, 48, 53]) has largely focused on programs that run atop embedded devices. Most of these approaches work by instrumenting the prover program so that at runtime, the instrumentation collects measurements that are periodically committed to the TEE, *e.g.,* the secure world of an ARM TrustZone-enabled chip on the embedded device. This measurement is signed by the TEE and provided to the verifier, who can then check whether the path followed by the program is the one expected for that program input. For example, CFLAT [2] adds instrumentation at the end of basic blocks of the program to compute a hash of the basic block, and integrates this hash into a cumulative accumulator of the execution path encountered so far. The accumulated hash is presented to the verifier as the path measurement, which then checks it using historical profiling data (*e.g.,* a database of expected hashes for each input). OAT [47] adds instrumentation after conditional instructions that commits the direction of the taken branch to the TEE as a bit-trace, together with the values of certain sensitive program elements (*e.g.,* return addresses and addresses of indirect jumps/calls). OAT's verifier uses symbolic execution of the program using bit-trace and recorded values to check the control-flow path taken by the prover.

Unfortunately, as we show in this paper, prior approaches do not scale up to *whole-program path attestation*, *i.e.,* to attest the entire control-flow path taken by the prover program. For example, OAT is tailored to attest the integrity of "operations" in embedded programs, which are specific regions of the program (manually

CCS '23, November 26–30, 2023, Copenhagen, Denmark.

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3616687

demarcated using annotations) that accomplish some well-defined task. The code that implements the movement of a robot hand would be demarcated as an operation, for instance. When the prover executes, the verifier only obtains an attestation measurement of the control-flow path pertaining to the operation, but not for the entire embedded program. Prior work has shown that the verifier can miss attacks directed against the vulnerabilities in the prover program when paths are attested only in certain parts of the prover's execution [28, Table IV]. Thus, it is desirable for a verifier to be able to attest the entire execution of the embedded program, rather than parts of it. Also, most prior work (*e.g.,* [2, 21, 39, 48]) has only been applied to small programs or those that work in specialized environments. When applied to these small programs or when path attestation is applied to record only a part of the program's behavior (*e.g.,* only "operations"), the corresponding program paths span only a few hundred or few thousand control-flow events. By avoiding whole-program path attestation, these methods are able to report a relatively modest runtime overhead on the prover's execution.

However, when we applied prior approaches to attest whole program paths in a suite of embedded benchmarks, we found that the approaches impose a prohibitive overhead on the runtime of the program (see Section 2). We show that the performance bottleneck stems from the sub-optimal way in which prior approaches instrument the prover program to measure the control-flow path taken, and that this approach fundamentally does not scale to whole-program path attestation. This instrumentation causes a large number of domain transitions to the TEE of the prover device (where measurements are stored securely), thereby imposing a heavy runtime overhead on the program.

We present **Blast**, an approach to scale attestation to whole program paths. Blast incorporates several new ideas based on the lessons learned from our detailed analysis of the overheads of prior approaches. It aims to reduce the number of TEE transitions that are used by prior approaches to record path attestation measurements (Section 3). Blast does so using *local logging—i.e.,* it stores measurements generated by the instrumentation in a log within the program's address space. Logging operations therefore do not require domain transitions and can be performed with a store instruction into the program's own address space. It isolates the log from the rest of the program by instrumenting the program to implement software fault isolation (SFI) [49]. Blast commits the log periodically to the TEE when the space allocated for the log is exhausted. Once committed, Blast reuses the log to continue recording path measurements.

Blast uses improved methods to record path measurements. We observe that prior approaches use sub-optimal ways to measure paths, which unnecessarily leads to more entries in the log. For example, CFLAT stores a path measurement at the end of every basic block while OAT does so at each branch. Such sub-optimal event recording fills up the log faster than necessary, which in turn triggers a TEE transition to commit the log. Blast instead adapts the idea of *Ball-Larus numbering* [10] from the program profiling literature. For an acyclic control-flow graph (CFG) of a function with N paths, Ball-Larus numbering selectively instruments edges of the CFG so that they together compute an integer in the range $[0, N-1]$ at function exit. This integer serves as a unique identifier of the path taken through the CFG. Ball-Larus numbering also supports CFGs

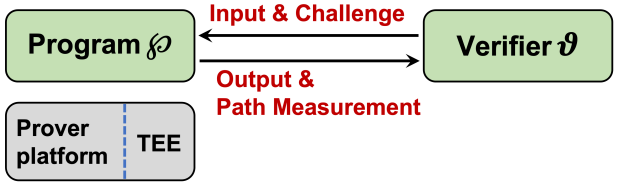


Figure 1: Basic setup for path attestation.

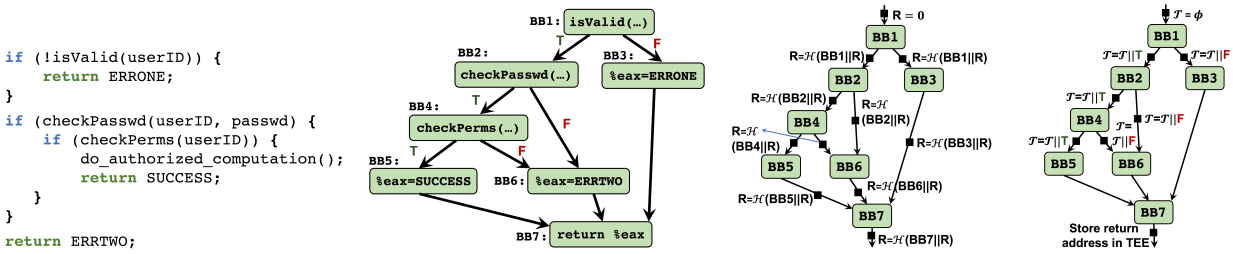
with loops; details in Section 3. Ball-Larus numbering is provably better than bit-tracing or other approaches to instrumentation in that it places instrumentation optimally, thereby resulting in lower runtime performance overheads than other approaches. Ball-Larus numbering also affords the flexibility of placing instrumentation so that the number of instrumented CFG edges along heavily-executed paths in the CFG is minimized. Using the Ball-Larus numbering approach, Blast reduces instrumentation encountered at runtime compared to both CFLAT and OAT. This in turn reduces the frequency at which Blast needs to commit the log to the TEE.

Blast incorporates a compact yet expressive representation of the recorded path measurement. Ball-Larus numbering records path numbers at a per-procedure level. Larus [34] extended Ball-Larus numbering to store whole-program path profiles using a grammar-based representation. In this approach, the log of Ball-Larus path numbers encountered is compressed to produce a compact context-free grammar that represents the whole program path. Although compact, the context-free grammar suffices to precisely reconstruct the entire path through the program. Blast offers the option of presenting this *context-free grammar based representation of whole program paths* to the verifier. The verifier can use this to easily reconstruct the precise path taken by the prover to process the input. Blast also supports the option of simply storing a hash-based commitment of the log, similar to the approach used by CFLAT. This option simply presents the verifier with a single hash-based measurement of the program path, which is checked against a database of acceptable values.

Our results show that Blast brings whole-program control-flow attestation to the realm of feasibility. On a set of embedded benchmarks (Embench-IOT [23]), Blast is able to attest whole-program control-flow paths while imposing an average runtime overhead of 67%. This is significantly smaller than the overheads of competing path attestation approaches when applied to whole program paths, which can slow the execution of the program by up to 100×-1000×.

2 BACKGROUND

Figure 1 shows the basic setup for path attestation. We consider a program $\mathcal{P}$, running on a prover platform, that is equipped with a TEE capable of attestation. A remote verifier, $\mathcal{V}$, provides an input to $\mathcal{P}$ together with a challenge (*e.g.,* a nonce), in response to which the prover platform provides the output of $\mathcal{P}$ on the input, and a path measurement in response to the challenge. The verifier $\mathcal{V}$ has local access to the program $\mathcal{P}$, but wishes to obtain a commitment from $\mathcal{P}$ on the control-flow path that it took to process the input. The prover platform enables this by collecting path measurements from $\mathcal{P}$ and uses the TEE to store the measurements securely. The prover then requests the TEE to digitally sign the measurements (incorporating the supplied challenge) and provides them to the



2(a) Code snippet of program $\mathcal{P}$. 2(b) CFG of Figure 2(a). 2(c) Instrumented by CFLAT. 2(d) Instrumented by OAT.

Figure 2: (a) Code snippet of running example program $\mathcal{P}$, (b) its corresponding control-flow graph (CFG), (c) CFG instrumented using CFLAT [2], and (d) CFG instrumented using OAT [47]. CFG edges with the black squares have instrumentation that domain switch into the TEE.

verifier in response. In this paper, we tailor our discussion and the prototype implementation of Blast to the ARM platform, with the TrustZone secure world as the TEE. However, the new ideas introduced in Blast are portable to any prover platform and TEE.

The verifier $\mathcal{V}$ checks the freshness of the signed measurements and then determines whether the control-flow path to which $\mathcal{P}$ has committed in the measurement is acceptable for that input. The precise details of the method that $\mathcal{V}$ uses to make this determination depend upon the amount of information available in the measurement sent by $\mathcal{P}$. For example, CFLAT [2] collects a set of cumulative hashes that uniquely identify the control-flow path followed in $\mathcal{P}$ as it processed the input. The verifier $\mathcal{V}$ can replay the input locally on $\mathcal{P}$ and check that hash values match. Alternatively, it can consult a local database that is pre-populated with acceptable hash values for program paths to process variety of input values.

In OAT [47], the attested measurement sent by the prover contains the direction of each conditional branch, and address of each indirect call or jump in $\mathcal{P}$, which $\mathcal{V}$ uses together with abstract execution (akin to symbolic execution) of $\mathcal{P}$ to determine the control-flow path followed in $\mathcal{P}$. However, as mentioned before, OAT does not attest whole program paths, and only records these measurements for a portion of the actual control-flow path in $\mathcal{P}$, namely, the parts of the path manually annotated as "operations".

2.1 Threat Model

All the approaches discussed in this paper rely on instrumenting $\mathcal{P}$ to collect path measurements at runtime. Blast relies on a compiler pass to insert this instrumentation, but the compiler does not have to be trusted because the correctness of the inserted instrumentation can be verified via a simple linear pass over the resulting executable. The verifier $\mathcal{V}$ does not trust the prover platform, but expects it to be equipped with a TEE, which it trusts. The verifier $\mathcal{V}$ assumes that the prover program $\mathcal{P}$ can be hijacked by an adversary, *e.g.,* by feeding malicious inputs that exploit zero-day vulnerabilities in $\mathcal{P}$.

We assume that the prover platform has implemented standard data-execution prevention techniques that are available on almost all modern hardware platforms, to prevent code injection attacks in $\mathcal{P}$. Preventing code injection attacks is important because the inserted instrumentation is key to precisely computing the path measurement that the verifier $\mathcal{V}$ will check. Commodity operating systems generally provide data-execution prevention in concert with the hardware (*e.g.,* W⊕X). However, embedded programs often execute atop bare-metal hardware, *e.g.,* ARM Cortex-M microcontrollers. Prior work has developed methods for data-execution prevention even for bare-metal settings [19, 33]. Such work is orthogonal to the focus of this paper, and as in prior work [2, 47], we simply assume that the prover platform provides bare-metal data-execution prevention, and can attest to the verifier $\mathcal{V}$ via boot-time integrity measurements that the defense is enabled.

Blast targets ARM TrustZone-based prover platforms and its design does not require $\mathcal{V}$ to trust any software running outside the TEE (*i.e.,* the secure world). For a program $\mathcal{P}$ executing atop a bare-metal normal world (*i.e.,* the rich-execution environment, or REE, of an ARM TrustZone platform), we show that Blast securely records paths measurements. Our prototype implementation uses OPTEE [41] to ease communication with the TEE, and OPTEE requires an operating system in the normal world. As we outline in Section 3.6, Blast can also be adapted to work in non-bare-metal settings, provided the normal world operating system is modified to incorporate safeguards that let the TEE protect Blast's security-critical state. Because the need for a normal world operating system is not germane to Blast's design, we chose not to incorporate these modifications in our prototype's OPTEE normal world, and assume it is benign in intent.

Code-reuse attacks (*e.g.,* return-oriented [46] or jump-oriented programming [11, 15]), and attacks that maliciously modify program data [27] that alter the program path followed in $\mathcal{P}$ can readily be detected because the path measurement that is collected by the TEE in the prover will diverge from the value that $\mathcal{V}$ expects. Our focus in this paper is only on attacks that alter the control-flow path in $\mathcal{P}$. Attacks that modify the value of a sensitive variable without altering control-flow path can be detected using additional instrumentation to record their values (*e.g.,* as done in OAT), but such instrumentation is orthogonal to the discussion in this paper.

2.2 Prior Art in Path Attestation

We now describe prior work on path attestation with a focus on CFLAT and OAT as representative approaches. Other approaches [2, 3, 21, 29, 39, 47, 48, 53] use largely similar methods. We use the code snippet in Figure 2(a) as a running example denoting $\mathcal{P}$. Figure 2(b) shows the control-flow graph (CFG) of $\mathcal{P}$. Both CFLAT and OAT instrument the program $\mathcal{P}$ to collect control-flow path measurements. However, they differ in the information collected and in the kind of locations where instrumentation is inserted.

CFLAT instruments the CFG of each function to store a rolling hash of the code of the basic blocks encountered during the execution of the function. CFLAT's path measurement is a set of hash values for each function encountered during $\mathcal{P}$'s execution.

For example, if the execution follows the path BB1→BB3→BB7 in the CFG shown in Figure 2(b), CFLAT records the hash value $\mathcal{H}(\text{BB7}||\mathcal{H}(\text{BB3}||\mathcal{H}(\text{BB1}||0)))$, where $\mathcal{H}(.)$ is a suitable collision-resistant hash function (BLAKE-2 in case of CFLAT) and || is a suitable concatenation operator.

Figure 2(c) shows how CFLAT instruments the CFG to compute the above hash value at runtime—each rectangle on an edge denotes that instrumentation is added to the edge. CFLAT maintains the path measurement in the TEE, and each instrumentation contains a trampoline that performs a domain switch into the TEE—a *world switch* to the secure world in ARM TrustZone using the `smc` instruction. At each such domain switch, CFLAT incorporates the hash of the preceding basic block into the path measurement that is maintained in the TEE, and transfers control back to $\mathcal{P}$ to resume execution (*i.e.,* a world switch into the normal world in ARM TrustZone). On entry to each function, the instrumentation initializes the hash value to 0. At each edge of the CFG, the instrumentation computes the hash of the basic block that executed preceding that hash, and appends the value to a per-function rolling hash maintained during the execution of that function. At function exit, the accumulated hash value(s) is reported as the path measurement of the function. A loop-free function that does not call any other functions will accumulate a single hash value, denoting the path taken by the function as it processed the input.

CFLAT supports functions that have loops or call other functions, by storing a set of hash values per function. To handle loops, CFLAT adds instrumentation on back-edges that saves the hash value computed thus far to the TEE, and re-initializes the hash computation to zero for the next iteration of the loop. Thus, CFLAT performs a domain switch at each loop header and collects a set of accumulated hash values, each accumulated hash corresponding to the execution of one acyclic fragment of the CFG. The number of accumulated hash values in the set also provides information (to $\mathcal{V}$) about the number of times each loop iterated. To support function calls, CFLAT's instrumentation saves and restores the value of the hash across call instructions encountered in a function. CFLAT adds instrumentation to save to the TEE the accumulated hash preceding a call instruction. When the called function returns, it adds instrumentation that restores the saved value from the TEE and resumes execution within the caller.

CFLAT reports path measurements on a per-function basis. A verifier presented with CFLAT's path measurements simply performs a lookup in a measurement database to check whether the computed hash values corresponds to the hash values of (one of) the acceptable path(s) for the input provided to the program. CFLAT assumes that the verifier has access to such a measurement database that is pre-populated with a list of acceptable hash values, *e.g.,* by profiling the program with various regression tests under a non-adversarial environment. In the absence of an entry in this database for a particular input, CFLAT's verifier must re-execute a local copy of $\mathcal{P}$ on that input, compute the path measurement, and check that it matches the value provided by the prover.

OAT requires application developers to annotate $\mathcal{P}$ to demarcate "operations." OAT's goal is to check the integrity of these operations via attestation. Each operation consists of a top-level function that can itself invoke other functions. However, in OAT, one operation must proceed to completion before another operation is invoked (*i.e.,* operations cannot nest). OAT reports one path measurement for each operation. The path measurement consists of: (a) a bit trace denoting the direction of conditional branches encountered during operation execution; (b) the return addresses of functions invoked during operation execution; and (c) addresses of functions invoked by indirect call instructions that were executed as the operation was performed. This path measurement is stored in the TEE and updated via the instrumentation that OAT inserts in $\mathcal{P}$.

CFG edge	CFLAT	OAT	Blast
Conditional branch	✓	✓	$\checkmark_{\text{LOG}}$ ‡
Unconditional branch	✓	✗	✗
Direct function call	✓	✗	$\checkmark_{\text{LOG}}$
Indirect function call	✓	✓	$\checkmark_{\text{LOG}}$
Return/function exit	✓	✓	$\checkmark_{\text{LOG}}$

‡In Blast, path measurements are committed to the local log only on conditional branches that are loop headers.

Table 1: Control-flow edges in CFLAT and OAT that contain instrumentation to perform a domain switch to the TEE. Although Blast also instruments some of these edges as shown in the table (with $\checkmark_{\text{LOG}}$), Blast's instrumentation stores the path measurement in a local log and does not trigger a domain switch (see Section 3).

To collect the bit-trace, OAT initializes the bit-trace when a new operation is started. At each control-flow edge following a conditional jump, OAT's instrumentation appends a single bit to the bit-trace, denoting the direction of the conditional. Figure 2(d) shows the CFG edges at which OAT adds instrumentation (this denotes a single function's CFG). Because the bit-trace is stored securely in the TEE, each instrumented edge will result in a domain switch to the TEE. The key difference from CFLAT is that OAT only instruments the edges following conditional statements, unlike CFLAT, where every CFG edge is instrumented. This, in turn, translates to fewer TEE switches at runtime, and therefore potentially lower runtime performance overheads. At the end of each function, OAT's instrumentation collects in the TEE the return address to which the present function returns (*i.e.,* the return address within the callee). OAT does not instrument direct call instructions because they unconditionally jump to the referenced function. However, OAT instruments indirect call sites to store the address of the called function in the TEE. OAT reports path measurements on a per-operation basis.

Table 1 summarizes the CFG locations instrumented by CFLAT and OAT, each of which involves a domain switch to the TEE. We have also shown Blast's instrumentation for comparison; as will be explained in Section 3, Blast's approach *does not* trigger a domain switch at these instrumentation locations.

2.3 TEE Domain Switches in Path Attestation

As must be clear from the preceding description, path attestation requires extensive instrumentation to be inserted into the program $\mathcal{P}$. Despite such intrusive instrumentation and the need for frequent domain switches, both CFLAT and OAT reported relatively low runtime overheads on $\mathcal{P}$'s execution. CFLAT was evaluated on a syringe pump application and the paper reports an absolute runtime overhead of a couple of seconds to collect path attestation measurements. OAT was applied to attest operation integrity of operations in five embedded applications, and reported an average runtime overhead on $\mathcal{P}$ of 2.7%.

Embench-IOT Program ↓	Number of Control-Flow Events (categorized by CFG edge types)					Total TEE Domain Switches Encountered at Runtime	
	Conditional Branches	Unconditional Branches	Loop Headers	Direct Calls	Returns and Exits	CFLAT	OAT
aha-mont64	384,507,002	456,417,002	189,927,000	8,460,006	8,460,006	**857,844,016**	**392,967,008**
crc32	174,420,002	348,670,002	174,250,000	174,420,006	174,420,006	**871,930,016**	**348,840,008**
cubic	660,007	970,003	310,000	200,006	200,006	**2,030,022**	**860,013**
edn	371,925,005	732,801,003	360,876,000	696,006	696,006	**1,106,118,020**	**372,621,011**
huffbench	495,825,002	486,255,002	233,266,000	1,078,006	1,078,006	**984,236,016**	**496,903,008**
malmult-int	406,732,884	794,099,724	387,366,840	92,807	92,807	**1,201,018,222**	**406,825,691**
minver	109,335,036	155,955,031	56,610,012	6,105,006	6,105,006	**277,500,079**	**115,440,042**
nbody	6,228,064	10,849,050	4,621,020	101,006	101,006	**17,279,126**	**6,329,070**
nettle-aes	78,000,771	147,732,515	51,168,256	858,006	858,006	**227,449,298**	**78,858,777**
nettle-sha256	30,400,019	185,250,019	24,225,008	3,800,006	3,800,006	**223,250,050**	**34,200,025**
primecount	880,205,002	726,973,002	282,281,000	1,006	1,006	**1,607,180,016**	**880,206,008**
sglib-combined	680,950,108	627,474,208	144,884,400	76,618,308	76,618,308	**1,461,660,932**	**757,568,416**
st	9,204,004	18,317,003	9,113,000	7,904,006	7,904,006	**43,329,019**	**17,108,010**
tarfind	80,905,424	114,040,424	48,400,474	36,331,006	36,331,006	**267,607,860**	**117,236,430**
ud	412,362,003	616,326,003	255,694,000	1,478,006	1,478,006	**1,031,644,018**	**413,840,009**

Table 2: Number of TEE domain switches seen at runtime with CFLAT's and OAT's instrumentation for the Embench-IOT benchmark suite. For all our experiments with Embench-IOT, we set `CPU_MHZ=1000`, a parameter denoting the number of runs of each top-level benchmark function. The total numbers for CFLAT and OAT can be computed from the control flow event types shown here, together with Table 1. This table also shows the number of branches that are loop header edges, which are the only branches at which Blast inserts log entries.

In an effort to understand CFLAT and OAT's performance when applied to attest whole program paths, we implemented their instrumentation approaches as compiler passes in LLVM-11.0.0, and instrumented benchmarks from the Embench-IOT suite [23]. Embench-IOT is a set of benchmarks that represents the requirements of modern connected embedded systems.

Table 2 reports the results of whole program path attestation on these benchmarks using the corresponding inputs provided as part of the suite. Recall that OAT requires developers to demarcate operations, whose integrity it attests; for the purposes of evaluation, we considered the whole program as one "operation." Because the raw runtimes depend heavily on the actual hardware platform used by the prover, for this part of our evaluation, we restricted ourselves to measuring the number of relevant control-flow events (classified using CFG edge types) encountered at runtime. We report raw runtimes with Blast on our hardware platform in Section 3.

For this experiment, we only instrumented direct call instructions and elided instrumenting indirect calls. Recall that OAT's approach instruments only conditional branches, unlike CFLAT. Note also that OAT avoids instrumenting direct call instructions (*cf.* Table 1), while they are instrumented by CFLAT. We merged the functionality implemented at function exits, and at the return point of the callee function into a single switch into the TEE, and Table 2 reports the consolidated number of TEE domain switches for that control-flow event.

To understand the raw performance implications of these numbers, recall from our prior discussion that both CFLAT and OAT make a TEE domain switch for *each* of the control-flow events for which they insert instrumentation. The time taken for a TEE domain switch differs based on the hardware platform, but OAT reported a TrustZone world switch time (switching to the TEE and returning) of 45μsec on their HiKey (with an ARM Cortex A53 processor) evaluation platform ([47, Table II]); CFLAT's evaluation also suggests an overhead of ~40μsec on their Raspberry Pi 2 evaluation platform ([2, Figure 8]). We used a Raspberry Pi 3 Model B+, for our evaluation of Blast, and observed an average world switch time of 190μsec on our platform. Observe that with such overheads for TEE domain switches and the numbers reported in Table 2, the raw runtimes for whole-program control-flow path attestation of the benchmarks in Embench-IOT will run into several hours. Given that the baseline execution time of these benchmarks is just a few seconds (under 35 seconds in all cases, *cf.*, Table 4), whole-program control-flow path attestation would impose *more than a 1000× overhead for most of these benchmarks*. We contrast this with Blast's results, where the overhead for Embench-IOT benchmarks is an average of just 1.67× (Section 4.1).

On further analysis, we noted that the benchmarks or "operations" measured in the experiments reported in both the CFLAT and OAT papers contain only a few thousand control-flow events. For example, the attestation reports in the five benchmarks to which OAT was applied were all *smaller than a kilobyte* in size. These attestation reports contain the bit-trace that records the direction of each conditional branch encountered, and addresses of indirect calls and jumps. This suggests that the operations in these benchmarks whose paths were measured consisted of *fewer than a thousand control-flow events*. This contrasts sharply with the numbers observed in whole program paths encountered in a modern embedded benchmark suite such as Embench-IOT (which denote the characteristics of a wide-variety of embedded applications), which contain hundreds of millions of control-flow events. We conclude that prior approaches to path attestation *do not scale to whole program path attestation*, thereby motivating us to develop Blast.

3 DESIGN AND IMPLEMENTATION OF BLAST

The analysis reported in the prior section shows that the key bottleneck that prevents prior methods from scaling to whole-program path attestation is the number of TEE domain transitions that they perform. Blast aims to reduce the number of TEE domain transitions encountered during path attestation by performing *local logging* of control-flow events (Section 3.1). That is, the instrumentation inserted by Blast writes out information to a local log in $\mathcal{P}$'s address space, rather than performing a domain transition on each control-flow event. The log accumulates a history of control-flow events, is periodically flushed to the TEE as it reaches capacity, and is reused to continue logging events until the program terminates.

However, this approach of local logging requires some care:

- *Optimizing events to be logged.* Local logging of control flow events can immediately improve the performance of prior approaches by reducing the number of TEE domain transitions. However, prior approaches are sub-optimal in the set of control-flow events that they instrument. Blast uses Ball-Larus numbering [10], an approach to optimally place instrumentation in the control-flow graph, to reduce the number of entries that are logged. This in turn reduces the rate at which the log is populated, and therefore the number of

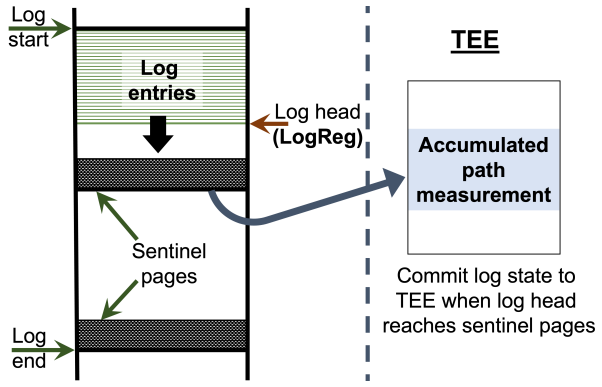


Figure 3: Structure of the log in Blast.

domain transitions required to commit the log to the TEE when it reaches capacity (Section 3.2).

• *Protecting log integrity.* Because the log is stored in the $\mathcal{P}$'s address space, it is vulnerable to attacks launched on the prover platform. Such attacks can alter the log state, and the resulting path measurement presented to $\mathcal{V}$ will no longer faithfully represent $\mathcal{P}$'s behavior on the prover platform. Blast protects the integrity of the log using software-fault isolation in $\mathcal{P}$ (Section 3.3).

Blast processes the log to present a commitment of the whole-program control-flow path followed by $\mathcal{P}$ to $\mathcal{V}$ (Section 3.4). It creates a compact grammar-based representation of the log [34], which can be presented to $\mathcal{V}$ to unambiguously reconstruct the path followed within $\mathcal{P}$. Finally, we qualitatively analyze the security of Blast (Section 3.6).

3.1 Avoiding TEE Domain Switches

In Blast, control events are committed to a log that is allocated locally within $\mathcal{P}$'s own address space. This avoids TEE domain switches on control events that contribute to the overhead of prior systems. Both CFLAT and OAT can be adapted to commit path measurements to a log rather than to the TEE.

Figure 3 shows how Blast structures the log. The memory required for the log is pre-allocated at the start of $\mathcal{P}$'s execution. On an ARM TrustZone system, Blast sets up the log from the TEE (*i.e.,* secure world) as a memory region that is shared between the TEE and the normal world. The program $\mathcal{P}$ executes in the normal world and writes entries into the shared region, and the log is readily accessible to the TEE as well. The size of the log is decided at the time when $\mathcal{P}$ is instrumented, and is aligned to start at a page boundary; both these requirements allow us to ease the instrumentation required to protect the log (in Section 3.3).

Each control event simply commits the relevant information to the log and increments the log header. The address of the log header is stored in a dedicated register that we call LogReg. Blast is implemented as an LLVM compiler pass, and can easily ensure that LogReg is reserved for exclusive use as the pointer to the log header by making this register unavailable for general register allocation when $\mathcal{P}$ is compiled. We quantify the cost of dedicating an exclusive register for LogReg later in the paper (Figure 5).

Blast logically divides the log into two symmetric halves. When one half is completely filled with log entries, its state is committed to the TEE. The program $\mathcal{P}$ can continue to execute and populate the other half of the log even as the first half is committed. Provided sufficient hardware resources are available on the prover platform (*i.e.,* a second CPU core), the execution of $\mathcal{P}$ and the operation to commit the log state to the TEE can proceed in parallel.

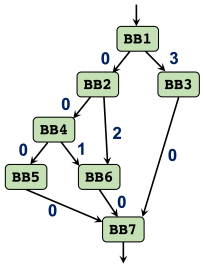
Blast's instrumentation to insert entries into the log simply adds the corresponding entry into the log and increments the value of LogReg. In particular, Blast does not perform any range checks on LogReg during each write, which would impose additional overhead on each store to the log. Instead, Blast uses the approach of inserting *sentinel pages*, one at the end of each half of the log, which are write-protected by the TEE. This approach ensures that a hardware fault is generated when LogReg points to a location in the sentinel page, indicating that the corresponding half of the log is full and that it is time to commit that half of the log to the TEE. The fault handler executes in the TEE and initiates the operation of committing the log state to the TEE in a separate thread. It changes LogReg in the main thread to point to the beginning of the other half of the log, and allows $\mathcal{P}$ to continue execution and generate log entries. Blast's fault handler write-protects the portion of the log that is being committed to the TEE, and makes this half writable only after the operation to commit it to the TEE is completed. This ensures that the log is not overwritten even if $\mathcal{P}$ exhausts the other half of the log even as the current half is committed to the TEE. We discuss the operations to commit the log into the TEE in more detail in Section 3.4.

3.2 Ball-Larus Numbering for Optimal Logging

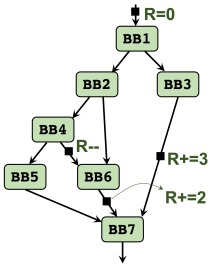
Blast builds upon the influential idea of Ball-Larus numbering [10]. The Ball-Larus algorithm places instrumentation in an optimal fashion by selectively adding instrumentation to the edges of a control-flow graph that increments the value of a counter in specific ways along specific edges. The instrumentation satisfies the invariant that the value accumulated in the counter will be between 0 and $N-1$, where N is the number of acyclic paths in the control-flow graph. The value in the counter uniquely determines the runtime path that was followed in the function. Ball-Larus numbering is provably up to 2× more compact than instrumentation for bit-tracing (*á la* OAT) [10, Section 2], and places instrumentation at fewer locations.

Background on the Ball-Larus Algorithm. The Ball-Larus algorithm works at a per-function level and adds instrumentation to the edges of the CFG of that function. The algorithm works in two steps: edge numbering, followed by edge instrumentation. We first describe the algorithm for acylic CFGs, and then generalize.

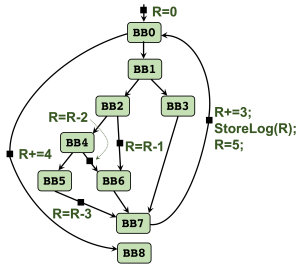
Given an acyclic CFG with uniquely-designated entry and exit nodes, the number of paths in the CFG is finite, say N. Ball-Larus numbering assigns a numeric value to each edge (valEdge(.)) of the CFG so that a runtime path from the entry to the exit node produces an integer identifier $\in [0, N-1]$ unique to that path. The identifier of the path is the sum of the valEdge(.) values of the edges in that path. Intuitively, the algorithm iterates over the edges of the CFG in reverse topologically-sorted order, and assigns to each node p a value valNode(p) that denotes the number of paths from p to the exit node. At each step, it assigns values valEdge(e) to each outgoing edge e from p so that the following invariant is maintained—the sum of the valEdge(.) values of the edges from p to the exit node is a unique integer $\in [0, \text{valNode}(p) - 1]$. Algorithm 1, adapted from



4(a) Ball-Larus numbering of CFG in Figure 2(b).



4(b) Instrumentation to compute path numbers (one alternative).



4(c) Instrumented CFG from Figure 2(b) with a loop added.

PathNum	Path
0	BB0→BB1→BB2→BB4→BB5→BB7
1	BB0→BB1→BB2→BB4→BB6→BB7
2	BB0→BB1→BB2→BB6→BB7
3	BB0→BB1→BB3→BB7
4	BB0→BB8
5	BB7→BB0→BB1→BB2→BB4→BB5→BB7
6	BB7→BB0→BB1→BB2→BB4→BB6→BB7
7	BB7→BB0→BB1→BB2→BB6→BB7
8	BB7→BB0→BB1→BB3→BB7
9	BB7→BB0→BB8

4(d) Path numbers for the CFG in Figure 4(c) obtained from the associated instrumentation.

Figure 4: Ball-Larus numbering and corresponding instrumentation. We also show how Ball-Larus numbering handles loops.

the original Ball-Larus paper [10], shows how valNode and valEdge are calculated for an acyclic CFG.

We are interested in the valEdge(.) values assigned to the edges of the CFG. Figure 4(a) shows one possible assignment of valEdge(.) values, for the reverse-topological ordering BB7, BB6, BB5, BB3, BB4, BB2, BB1 (there can be multiple assignments, based on how the nodes are sorted). Each path from the entry to the exit gets a unique value $\in [0, 3]$ as there are 4 paths in this CFG.

Algorithm 1: Ball-Larus acyclic CFG edge numbering.

Input: Control-flow graph (CFG) of a function
Output: valEdge($p \rightarrow q$) for each edge of the CFG

```
1  nodeList = nodes of CFG, in reverse topologically sorted order
2  for (each p ∈ nodeList) do
3      if (p is a leaf node) then
4          valNode(p) = 1
5      else
6          valNode(p) = 0
7          for (each edge p→q) do
8              valEdge(p→q) = valNode(p)
9              valNode(p) += valNode(q)
10         end
11     end
12 end
```

With edges numbered with valEdge(.) values, the instrumentation algorithm places instructions along edges that compute the value of each path. One obvious approach would be to simply increment the path counter using the valEdge(.) value of each edge. However, the Ball-Larus approach provides significant flexibility in how instrumentation is inserted. In particular, it allows weights to be assigned to edges, and the Ball-Larus approach places instrumentation so that the overall weight of the instrumented edges is minimized. It does so by identifying a maximum weight spanning tree of the CFG so that the weight of the chords (*i.e.,* the edges of the CFG that are omitted from the spanning tree) is minimized. It places the instrumentation on the chords. There can therefore be multiple ways in which to instrument the CFG, and the edge weights provide significant flexibility in minimizing instrumentation along hot paths. Figure 4(b) shows one way to instrument the CFG to compute path numbers.

The Ball-Larus algorithm generalizes to loops much like CFLAT. Each loop induces a back-edge in the graph. We insert instrumentation on back-edges that records the path number recorded until execution reaches that edge, and reset the path number on that edge, effectively breaking the CFG into acyclic portions. A small modification to the original Ball Larus numbering algorithm also yields unique path identifiers for the loop in its entirety as illustrated in Figure 4(c), which adds a loop to the CFGs from our running example. As can be seen, the Ball-Larus approach adds instrumentation to the back-edge BB7→BB0 to increment the value of BLReg computed thus far, store it in the log, and then reset the value of BLReg to 5. When reading the acyclic paths ending in BB7, the register operation BLReg+=3 must also be added to obtain the path number of the corresponding acyclic path (*i.e.,* the path number of the path is the value of BLReg just before it is committed to the log). This is akin to the final operation performed on the exit of a function with an acyclic CFG. The corresponding path numbers in this CFG for the acyclic portions are as shown in the adjoining table in the figure. We refer the reader to the original paper [10] for complete details on handling loops.

We have described the Ball-Larus algorithm at a per-function level for a single-threaded program. Blast records whole program paths by storing the path numbers taken in each function encountered along the path. In Section 3.4, we describe Blast's compact representation of the whole program path that builds on prior work in this area [34]. The Ball-Larus algorithm can also soundly record path numbers in multi-threaded programs (path numbers are recorded per thread), but our Blast prototype currently works for single-threaded programs. CFLAT and OAT's approaches do not work on multi-threaded programs as presented.

Ball-Larus Instrumentation for Logging. Blast directly leverages Ball-Larus instrumentation placement. In Blast, we dedicate a register (BLReg) that computes the path number at runtime. On entry into a function's CFG, Blast's instrumentation initializes BLReg to 0. It adds operations to increment or decrement BLReg to the CFG's edges as determined by the Ball-Larus algorithm (*e.g.,* Figure 4(b)). At the exit edge, Blast adds instrumentation to store the value of BLReg to the log. Thus, for an acyclic CFG that does not invoke any other functions (*i.e.,* a call-graph leaf node), *Blast appends only a single entry to the log*. Loopy CFGs and function calls require additional log entries, as will be explained.

It is important to note a key point of difference between CFLAT, OAT, and Blast's approaches. Note that in both CFLAT and OAT the inserted instrumentation performs a domain switch to store path measurements in the TEE. Even if CFLAT and OAT were modified to use local logging, each instrumented CFG edge would be a control-flow event that must be appended to the log. In contrast, Blast computes the path number via register operations to BLReg for the most part, and only commits the value of BLReg at certain CFG locations (at function exits, calls to other functions, and loop

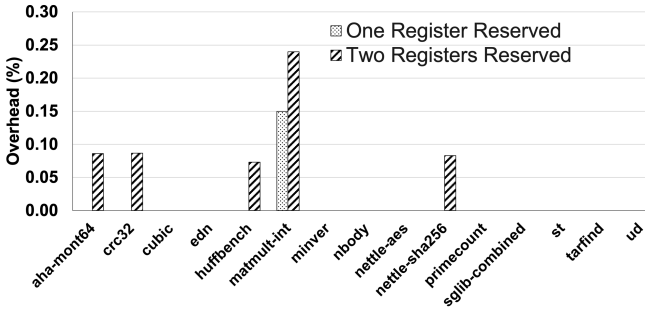


Figure 5: Impact of reserving registers for BLReg and LogReg.

back edges). This also means that Blast must protect the path number being computed in BLReg from malicious modifications until it is committed to the log. The register BLReg must also not be overwritten by other benign operations in $\mathcal{P}$.

To protect BLReg from attack and to prevent it from being overwritten, we implement Blast's instrumentation as an LLVM compiler pass, and remove BLReg from the pool of available registers for general register allocation for $\mathcal{P}$ (just as it did with LogReg). Thus, the only occurrence of BLReg and LogReg is at instrumentation locations, and the contents of BLReg and LogReg are not spilled to or stored in $\mathcal{P}$'s memory. Because we assume that the prover runs $\mathcal{P}$ with data-execution prevention enabled, the attacker is also unable to insert code that can otherwise modify BLReg and LogReg. That leaves code-reuse attacks as the only remaining threat vector. However, even such attacks are readily detected by Blast if they alter the control-flow path in $\mathcal{P}$ (see Section 3.6).

Blast's approach requires two registers—BLReg and LogReg—to be exclusively reserved for the required instrumentation. In the absence of these registers, operations that could otherwise use these two additional registers would instead have to be implemented with memory operations instead, which leads to runtime overhead. Figure 5 quantifies the impact of register reservation for the Embench-IOT benchmark suite. We ran these experiments on a Raspberry Pi 3 Model B+ platform with an ARM Cortex A53 Quad core 64-bit processor clocked at 1.4GHz, equipped with 1GB LPDDR2 SRAM. We compared the performance of the baseline (*i.e.,* the benchmark executing with all available registers) both with a single register reserved, as well as with two registers reserved. As this figure shows, except for `matmult-int`, the performance of the remaining benchmarks remains unaffected if a single register is reserved. If two registers are reserved for instrumentation, as required in Blast, more benchmarks are impacted, but in all cases, the runtime overhead remains under 0.25%. While it is true that reserving registers will in general result in runtime performance overhead, we can conclude that at least for the Embench-IOT suite, the impact of reserving registers is minimal.

We now discuss how Blast works on CFGs with loops and function calls. Loops introduce cycles in the CFG and therefore lead to an unbounded number of paths. Blast uses Ball-Larus' trick of resetting the path number (to a non-zero value) at the back-edge of the loop (*i.e.,* the entry point into the loop header) to split the CFG into a set of acyclic components, each of which computes path numbers independently. Because the path number is reset at loop entry, Blast saves the value of BLReg into the log. Thus, for a loopy CFG, the integrity measurement of a function is a set of path numbers, each denoting the execution of one acyclic component of the CFG. The path number at the loop header is reset in such a way that the the resulting path numbers of each acyclic component are unique across the function, and help identify the acyclic component being executed. A fresh measurement is stored in the log for each loop iteration, thereby also helping identify the number of times a loop iterated. See Figure 4(c)/(d) for an example.

Because Blast records path numbers at function granularity, a function call (direct or indirect) requires a reset of the path counter in BLReg to store the value of the path counter. In Blast, we insert instrumentation at function call instructions to record the address of the called function in the log, and also commit the current value of BLReg to the log, so that it can be reset for use in the called function. As with our handling of loops, this also requires a slight redefinition of how path numbers are computed. In particular, CFG edges leading into a basic block containing a procedure call terminate acyclic paths (as also discussed in prior work [34]), and path numbers are computed afresh for subsequent segments of the control-flow graph of the function. The called function resets BLReg and computes a path number, which is similarly committed to the log. Following the call instruction, BLReg is initialized to suitably so that the following acylic path fragment in the CFG also yields a unique path number within that function (as with loops). The code snippet below shows an example of Blast's instrumentation inserted before a function call (`check_alarm` is the called function, shown in bold). Blast uses ARM64 register `x19` for LogReg and `w20` for BLReg:

```
str w20, [x19], #4              // store path value in log before call
mov w8, #func_id                // store ID of called function in log
str w8, [x19], #4               // (continuation of above step)
bl func_addr <check_alarm>      // function call
mov w20, #init_val              // reset BLReg as per Ball-Larus
```

To summarize, the integrity measurement recorded during the execution of $\mathcal{P}$ includes, for each function: (a) the full set of acyclic path numbers encountered within the function; (b) addresses of functions called by that function (both directly and indirectly); (c) addresses of the locations in the caller to which called functions return. This information is available for each function. The last column in Table 1 qualitatively compares the instrumentation in Blast to CFLAT and OAT.

Empirically, we found that Ball-Larus numbering results in many fewer control events requiring entries in the log when compared to CFLAT and OAT adapted for local logging (instead of TEE domain switches to store measurements). Table 4(c) shows that Blast outperforms CFLAT and OAT even when they are adapted for local logging. In this section, we show that this is empirically because Blast's Ball-Larus approach results in many fewer control events requiring entries in the log when compared to CFLAT and OAT adapted for local logging (instead of TEE domain switches to store measurements). Table 3 shows the total number of log entries that result using Blast's approach for each of the Embench-IOT benchmarks, and also how this compares to the corresponding number of log entries in a log-based adaptation of CFLAT and OAT. Recall that Blast creates log entries only for loop headers, function calls, and returns/function exits (*cf.* Table 1). The number of log entries reported in Table 3 is therefore simply the sum of the corresponding control events reported in Table 2.

Embench-IOT Program ↓	# Log entries using Blast's approach	CFLAT / Blast	OAT / Blast
aha-mont64	206,847,012	**4.14**×	**1.90**×
crc32	523,090,012	**1.66**×	**0.66**×
cubic	710,012	**2.85**×	**1.21**×
edn	362,268,012	**3.95**×	**1.03**×
huffbench	235,422,012	**4.18**×	**2.11**×
malmult-int	387,552,454	**3.09**×	**1.05**×
minver	68,820,024	**4.03**×	**1.68**×
nbody	4,823,032	**3.58**×	**1.31**×
nettle-aes	52,884,268	**4.30**×	**1.49**×
nettle-sha256	31,825,020	**7.01**×	**1.07**×
primecount	282,283,012	**5.69**×	**3.18**×
sglib-combined	298,121,016	**4.90**×	**2.54**×
st	24,921,012	**1.74**×	**0.68**×
tarfind	121,062,486	**2.21**×	**0.97**×
ud	258,650,012	**2.21**×	**1.60**×

Table 3: Number of control-flow events with Blast's instrumentation that result in entries into the local log. The last two columns show how many fewer entries the Ball-Larus instrumentation approach inserts into the log compared to the approaches used by CFLAT and OAT (reported as TEE domain switches in Table 2).

As Table 3 shows, Blast reduces the number of log entries by up to 7× compared to CFLAT, and up to 3.18× compared to OAT. For a handful of benchmarks (`crc32`, `st`, and `tarfind`), we observed that Blast results in more log entries than a log-based adaptation of OAT. Upon further analysis, we found that this was because these benchmarks contain a large number of (direct) function calls. Recall (from Table 1) that direct function calls are uninstrumented in OAT, but do require instrumentation in Blast to commit the current value of `BLReg` to the log. In these cases, inlining can help Blast significantly reduce the number of control-events that can be logged by eliminating direct function calls. For example, inlining function calls in `crc32`, a benchmark in which there are a large number of direct function calls to small functions, completely eliminates the need to insert log entries into direct calls, thereby resulting in a total of 348,670,006 log entries, which marginally improves upon OAT (348,840,008 entries, as reported in Table 2). Section 4 presents detailed results showing the impact of inlining.

3.3 Log Integrity using Software Fault Isolation

Blast must protect the integrity of the log from unauthorized writes. In particular, only the instructions inserted as part of Blast's instrumentation are allowed to append entries into the log. While Blast uses register reservation to ensure that `LogReg` cannot be modified in $\mathcal{P}$, it must also ensure that memory store instructions in $\mathcal{P}$ do not target the log region.

As mentioned in Section 3.1, Blast sets up the log as a memory region that is shared between the secure world and normal world of an ARM TrustZone prover platform. The size of the log is fixed prior to instrumenting $\mathcal{P}$. Because the size of the log is known, the instrumentation can hard-code this size in the fault isolation range checks that Blast's instrumentation inserts in $\mathcal{P}$. The log is page aligned so that sentinel pages cleanly represent the boundaries of the log halves. Blast's implementation uses the `TEEC_AllocateSharedMemory` from the OPTEE library [41] that helps set up shared memory regions that are cleanly aligned with page boundaries. Blast stores and write-protects the start address of the log in the global area so that it is not modified during $\mathcal{P}$'s execution.

After every memory store instruction, Blast adds instrumentation to fetch the address of the store, retrieves the start address of the log, and ensures that the store address does not target the log. The following snippet shows the instrumentation inserted after a store instruction (shown in bold) assuming a 1MB log.

```
str w8, [x29, #4]                    // perform the store
add x9, x29, #4                      // obtain the store address
and x9, x9, #0x7fffffffff00000       // mask the store address
ldr x10, log_start_addr              // 1MB-aligned log start
subs x9, x9, x10                     // if equal, then abort.
b.e _abort
```

The store instruction in this snippet writes the memory address `[x29+4]` (`x29` is an ARM64 register). We ensure in the shared memory allocator (from `TEEC_AllocateSharedMemory` that creates the log) that the log starts at a 1MB aligned address for a 1MB log, which allows us to implement the fault isolation check using just a mask instruction rather than a range check. The instrumentation masks the memory address of the store on the third line (in `x9`), and obtains the 1MB aligned start address of the log in `x10`. The masked value in `x9` will be equal to the value in `x10` only if the store address pointed to a location within the log; this in turn aborts execution.

Unlike traditional SFI [49], we insert the check *after* the store. If the check was placed before the store, then it may be possible for a control-flow integrity attack [1] on $\mathcal{P}$ to bypass the preceding SFI check and directly transfer control to the store instruction. In our approach, the store will necessarily be followed by the SFI check, thereby aborting execution if the store address targets the log.

3.4 Path Measurement and Verification

Blast's fault handler is triggered when one half of the log is filled to capacity (via the write to the sentinel page). This fault handler creates a new thread that executes in the TEE and begins the operation of committing the log to the TEE. We note that the log itself need not be copied into the TEE—it suffices to obtain a commitment of its state. For example, it suffices to store a hash of the contents of the log in the TEE. If required, the log's contents can be saved within the untrusted partition, *e.g.,* the normal world of an ARM TrustZone platform. Its integrity can be checked at any time using the hash value committed to the TEE. Blast stores a *single* hash value as the commitment of the log. When the next half of the log is full, its hash is integrated with the value already in the TEE, thereby resulting in a cumulative hash.

Recall that the log is set up in memory that is shared between the TEE and the normal world. The thread that is tasked with committing the log's state to the TEE can work concurrently with the thread that continues $\mathcal{P}$'s execution. Note that the former thread must perform a domain switch (*e.g.,* via an `smc`, switching the processor into the secure world), while the latter thread executes in the normal world. On a multicore prover platform, the thread that commits the log state to the TEE can execute in parallel with $\mathcal{P}$, thus eliminating log commitment cost from $\mathcal{P}$'s execution.

The prover can present (a signed, nonced) hash that serves as the log commitment to $\mathcal{V}$. Provided that $\mathcal{V}$ has a measurement database of acceptable hash values, indexed by input (as in CFLAT [2]), this hash value itself serves as a commitment from the prover to $\mathcal{V}$ on the control-flow path followed within $\mathcal{P}$. However, there are several situations in which the hash value by itself proves insufficient for $\mathcal{V}$ to learn about the path followed in $\mathcal{P}$. As examples, (1) $\mathcal{V}$ may not have an entry in its measurement database of the acceptable hash value for a particular input; (2) $\mathcal{P}$'s execution may involve executions of loops in its code, and the number of iterations is not known to $\mathcal{V}$ *a priori*; or (3) $\mathcal{P}$'s execution may be non-deterministic

Log entries	Id
<foobar, 2>	↦ a
<foo, 8>	↦ b
<bar, 9>	↦ c
<foobar, 5>	↦ d
<foo, 8>	↦ b
<bar, 9>	↦ c
.....	

WPP grammar:

S → aXdX
X → bc

Each line is a production rule in the context-free grammar, S and X are the non-terminals output by the WPP generation algorithm and the lower-case symbols are terminal symbols.

Figure 6: Fragment of a log produced by Blast. Each entry is a tuple of the form <func_name, *Ball_Larus_PathID*>. Each unique entry is mapped to a distinct terminal symbol to represent it in the WPP.

because of signal handlers in its code that are triggered during $\mathcal{P}$'s execution, or because $\mathcal{P}$ is multi-threaded.

In each of these cases, presenting $\mathcal{V}$ with the full log of control-flow events allows it to reconstruct the execution of $\mathcal{P}$ step by step. The hash received from the prover's TEE serves as the commitment of the log's state, and $\mathcal{V}$ can verify that the hash indeed corresponds to the log that it receives from the prover.

However, the key challenge is that the log itself can be very large, and this can result in the prover having to transfer several megabytes of data to $\mathcal{V}$. Blast draws upon a classic idea from the program profiling literature—that of compactly representing a log of the program's execution using a context-free grammar, called the *WPP* [34]. WPPs build upon Ball-Larus numbering, and compactly represent a single control flow path through the whole program, including calls and returns from functions (with acyclic path fragments represented in the WPP using its Ball-Larus number), as well as precisely capturing loop iterations.

WPP construction from a log of control-events proceeds as follows. As collected, each entry in the log generated by Blast can be thought of as a tuple <func, *PathID*>, that represents the acyclic Ball-Larus path number *PathID* followed within the function name `func`. Each such tuple is assigned an identifier that then becomes a terminal symbol in the resulting grammar. The WPP construction algorithm [34] identifies common fragments in the log, and "lifts" them to non-terminal symbols. Consider, for example, a fragment from a log shown in Figure 6. This log fragment can be represented symbolically as abcdbc, and its compressed WPP representation would be as shown on the right in Figure 6. Intuitively, the WPP construcion algorithm identifies repeated sequences of control-events that appear in the log (*e.g.,* generated by execution of loops or repeated invocations of functions) and compresses their occurrence into a non-terminal symbol. The WPP grammar satisfies two properties: (1) it is a more compact representation of the "string" representation of the log (*i.e.,* the string obtained by composing the terminal symbol identifiers assigned to each <func, *PathID*> tupe); and (2) the log is the only "string" that can be generated from the starting non-terminal symbol of this context-free grammar (S). We refer the reader to the original paper [34] for full details on the algorithm to construct the WPP.

If the value presented by the prover does not suffice (for any of the reasons outlined earlier), $\mathcal{V}$ can request the prover send it the WPP representation of the log. The prover can run the WPP construction algorithm offline on the log, and send the WPP to $\mathcal{V}$. The verifier can independently reconstruct the log from the WPP, check that the hash it received from the prover's TEE can be obtained from this log, and then uses the log to identify the control-flow path followed in $\mathcal{P}$. Figure 7 summarizes the workflow for path measurement verification.

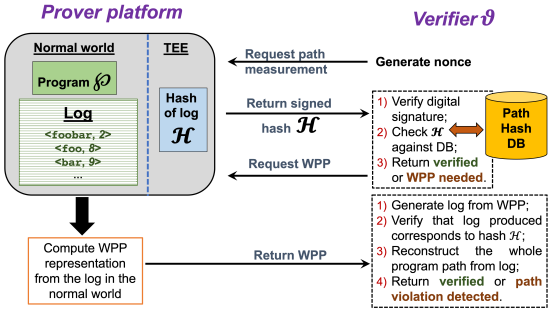


Figure 7: Workflow for verification of path attestation.

3.5 Implementation

We have implemented Blast's instrumentation as an LLVM (version 11.0.0) compiler pass that uses `LogReg` and `BLReg` exclusively to store a pointer to the log, and to accumulate path counter information, respectively. As it instruments $\mathcal{P}$, it also emits the list of program locations where instrumentation is inserted. Although we trust the compiler to instrument the $\mathcal{P}$ correctly, compilers are complex and could be buggy. The list of program locations emitted by the compiler allows us to build a simple static binary analyzer that performs a linear pass over $\mathcal{P}$ to ensure that `LogReg` and `BLReg` are used exclusively at the instrumentation points (*i.e.,* that it does not appear anywhere else in the program's text). In practice, we target the ARM-64 bit architecture. We use the register `x19` as `LogReg` and `w20` as `BLReg`. We therefore disallow these registers (and also `w19`, which are the 32 LSB bits of `x19` and `x20`, whose 32 LSB bits `w20` represents) for general-purpose register allocation during compilation. Our implementation uses the ARM TrustZone as the TEE, and we link $\mathcal{P}$ with the OPTEE library [41] to ease domain switching when the log state must be committed to the TEE. This allows us to include just the logically simple static analyzer in the TCB, which can check the work of the benign (but potentially buggy) program instrumenter. Our prototype uses the Blake2s function to hash the log and we configured it to produce a 32-byte hash. Because of the associated engineering overheads, we have built the Blast prototype to only supports single-threaded programs, although Blast's instrumentation and verification approaches readily extend to multi-threaded programs.

3.6 Qualitative Security Analysis

In Blast, an attacker is a malicious prover that alters the execution of $\mathcal{P}$ while ensuring that the path measurement reflects a "correct" execution to $\mathcal{V}$. We show that Blast's design makes this task difficult for the attacker. Code corruption attacks that simply replace $\mathcal{P}$'s code are easily detected because the path numbers and addresses of function calls/return addresses collected in the path measurement will not match. We assume that the prover platform implements data-execution prevention in the normal world to eliminate code injection attacks on $\mathcal{P}$. With Blast's SFI instrumentation, this ensures that any memory stores from instructions that are already in $\mathcal{P}$ do not impact the integrity of the log.

We are thus left with the possibility of the attacker launching code-reuse attacks on $\mathcal{P}$. Because Blast's goal is to faithfully record the program path followed in $\mathcal{P}$, a successful attack would have to cause the path measurement recorded in the log to deviate from

the actual path followed in $\mathcal{P}$. Thus, the attack must not leave any trace in the log that will be visible to $\mathcal{V}$. For this to be possible, the attacker must either (a) modify BLReg suitably (to reflect a different path number than the one actually followed) between two control-flow events that result in entries being appended to the log; or (b) violate the append-only property of the log by modifying LogReg to point to an earlier fragment of the log that will then cause any log entries inserted during the attack to be overwritten. We argue that neither case is possible.

For case (a) to happen, the attacker must identify and chain together gadgets that modify the BLReg register suitably. Recall that Blast reserves BLReg for exclusive use at instrumentation locations, and this register cannot be used elsewhere in the program. Thus, the gadgets will have to be composed using instrumentation inserted by Blast. However, any attempt in the attack to chain together gadgets using non-local control-flow instructions (such as returns or jumps) will result in a log entry because Blast inserts log entries at such locations (Table 1). Moreover, Blast's instrumentation at all return or indirect jump locations records the corresponding return/jump addresses in the log. For case (b), the attack must reset the value of LogReg to point to an earlier fragment of the log. Blast reserves LogReg for exclusive use at instrumentation locations, and every such location only increments LogReg. Its value is only reset within Blast's fault handler, but the fault handler also causes the log to be committed to the TEE. Thus, the attacker cannot reset LogReg without leaving a detectable trace. Although Blast can generalize to any architecture, our prototype currently targets ARM64, a RISC architecture with fixed-length instructions that start at word-aligned boundaries, and for which gadget construction is more challenging than on the x86 [12, 20].

The discussion above assumes that $\mathcal{P}$ executes atop a bare-metal normal world, and that the entire program $\mathcal{P}$ is instrumented with Blast, including any interrupt handlers in $\mathcal{P}$ (as was also assumed in OAT). However, the presence of an operating system in the untrusted normal world could undermine security. For example, the operating system could maliciously modify the stack-saved values of BLReg, LogReg or the log itself when it gets control via a system call or during interrupt/exception handling. Neto and Nunes [36] provide a detailed overview of the security implications of interrupt handling to path attestation. Prior work [9, 26] developed an approach that modifies the normal world operating system to allow security-sensitive operations to be mediated by the TEE, thus protecting the integrity of key normal world data structures (*e.g.,* page tables). Blast can build on this approach by additionally requiring control transfers from $\mathcal{P}$ to the operating system to be mediated by the TEE. The TEE saves the values of BLReg and LogReg, and computes a checksum over the log before allowing the normal world operating system to perform its task. Once the task is complete, the TEE restores BLReg, LogReg, and ensures using the checksum that the log is unmodified. The TEE must ensure via boot-time attestation that this modified operating system is loaded into the normal world and protect it from unauthorized modification [9, 26]. Running $\mathcal{P}$ atop such a normal world operating system will naturally result in additional TEE domain switches, and evaluating the cost of this approach is an interesting topic for future research.

Blast's focus is on control-flow path measurement, *i.e.,* to ensure that the verifier has an accurate picture of the control-flow path followed in $\mathcal{P}$. It does not aim to attest the integrity of data operations. Attacks or unauthorized modifications of program data that alter the control-flow path followed will still be detectable by $\mathcal{V}$ using Blast's path measurement. However, it may still be possible for an attacker to modify the value of a sensitive variable in $\mathcal{P}$ that does not alter the program path taken. Such attacks are not within the threat model or the scope of Blast, and can be addressed using additional instrumentation to also log the values of these sensitive variables in the integrity measurement [47].

4 EVALUATION

We evaluated Blast using the Embench-IOT benchmark suite atop a Raspberry Pi 3 Model B+ that runs an ARM Cortex A53 Quad-core 64-bit processor clocked at 1.4GHz and is equipped with 1GB LPDDR2 SRAM. In our evaluation:

① Table 4 presents the runtime overhead and memory overhead imposed by Blast's instrumentation on $\mathcal{P}$'s execution. It also compares Blast with CFLAT and OAT adapted to use local logging;

② Figure 8 breaks down the contribution of each of Ball-Larus instrumentation, SFI, and log commit operations to this overhead;

③ Table 5 analyzes the energy consumption of Blast;

④ Table 6 quantifies the benefits of WPP for log compression;

⑤ Section 4.4 shows the effectiveness of Blast at detecting anomalous behavior in $\mathcal{P}$ using a case study.

4.1 Performance Analysis

Recall that Blast's fault handler spawns a new thread to commit the log state to the TEE via hashing. This thread is logically concurrent with $\mathcal{P}$'s execution, and provided sufficient hardware resources are available (*i.e.,* an additional CPU core on which it can switch into the TEE and execute), $\mathcal{P}$'s execution and the log commit operation can execute in parallel. In this section, we first evaluate Blast assuming the presence of an additional core to periodically execute the commit thread. We then analyze Blast's performance for the case where an additional thread is not available, and the commit thread's execution interleaves with $\mathcal{P}$ on the same CPU core.

Performance with parallel log commit. Table 4(a) compares the execution time and the size of the instrumented binary of $\mathcal{P}$ with a baseline in which $\mathcal{P}$ runs without any instrumentation. The thread that commits the log executes on a separate CPU core than $\mathcal{P}$ and computes a rolling Blake2s hash of the log in the TEE. The log is a 1MB region pre-allocated in $\mathcal{P}$'s address space.

The first set of results (without function inlining) shows that Blast's instrumentation results in an average runtime overhead of 185% across all the benchmarks. The instrumented binary is 64% bigger than the uninstrumented version of $\mathcal{P}$—this metric is called bloat. While most of the benchmarks have an overhead of less than 100%, there was significant slowdown for a handful of benchmarks, namely `crc32` (808%), `sglib-combined` (215%), `st` (528%), and `tarfind` (518%). Upon further analysis, we found that these benchmarks have a large number of direct function calls. Blast commits the value of BLReg to the log at direct function calls, and for these benchmarks, the log is filled faster than the thread that commits its state to the TEE can complete execution. Thus, $\mathcal{P}$ is forced to wait until the log is committed before it can resume execution.

Embench-IOT Program ↓	Baseline		Blast w/o func. inlining		Blast with func. inlining		Best case ovhd.
	Time (sec)	Code (KBs)	Time (s) & Overhead	Code (KB) & Bloat	Time (s) & Overhead	Code (KB) & Bloat	
aha-mont64	11.62	15	29.15 (151%)	29 (93%)	**28.10 (141%)**	53 (253%)	141%
crc32	11.58	19	105.23 (808%)	29 (52%)	**18.33 (58%)**	33 (73%)	58%
cubic	1.77	115	2.04 (15%)	125 (8%)	**2.02 (14%)**	149 (29%)	14%
edn	26.03	23	**45.72 (75%)**	37 (60%)	55.88 (114%)	53 (130%)	75%
huffbench	13.72	19	**25.61 (86%)**	33 (73%)	26.01 (89%)	61 (221%)	86%
matmult-int	33.42	19	**44.21 (32%)**	29 (52%)	45.47 (36%)	37 (94%)	32%
minver	4.39	19	8.88 (102%)	29 (52%)	**7.96 (81%)**	65 (242%)	81%
nbody	0.52	15	**0.82 (57%)**	25 (66%)	0.84 (61%)	37 (146%)	57%
nettle-aes	16.47	32	21.57 (31%)	46 (43%)	**21.47 (30%)**	90 (181%)	30%
nettle-sha256	12.04	27	13.24 (9%)	37 (37%)	**13.14 (9%)**	45 (66%)	9%
primecount	15.34	14	**28.44 (85%)**	24 (71%)	28.48 (85%)	32 (128%)	85%
sglib-combined	16.84	39	53.11 (215%)	101 (158%)	**32.98 (95%)**	145 (271%)	95%
st	0.80	15	5.03 (528%)	25 (66%)	**1.47 (83%)**	49 (226%)	83%
tarfind	3.76	15	23.26 (518%)	25 (66%)	**6.89 (83%)**	41 (173%)	83%
ud	18.14	15	**30.90 (70%)**	25 (66%)	31.79 (75%)	45 (200%)	70%
Average Overhead:			185%	64%	70%	162%	**67%**

4(a) Log commit thread parallely executing with $\mathcal{P}$ on dedicated CPU core. We compare the runtime overhead/code bloat of Blast with and without function inlining and report the best case.

Blast w/o parallellism	
Time (s) & Overhead	Waiting Time (s)
46.10 (296%)	18.5
35.05 (202%)	17.13
2.05 (15%)	0.03
80.51 (209%)	35.81
48.44 (253%)	23.58
81.51 (143%)	38.01
13.31 (203%)	5.6
1.26 (142%)	0.45
26.37 (60%)	5.05
16.38 (36%)	3.42
50.6 (229%)	27.31
55.96 (232%)	23.61
2.32 (190%)	0.9
11.51 (206%)	4.74
56.03 (208%)	26.11
Avg. Ovhd.: 175%	

4(b) Log commit thread interleaved with $\mathcal{P}$.

Blast versus CFLAT & OAT			
CFLAT time (s)	CFLAT Blast	OAT time (s)	OAT Blast
87.12	(3.10×)	40.4	(1.44×)
105.23	(5.74×)	52.64	(2.87×)
2.08	(1.03×)	2.03	(1.00×)
111.24	(2.43×)	45.29	(0.99×)
99.2	(3.87×)	50.09	(1.96×)
120.68	(2.73×)	43.83	(0.99×)
28.61	(3.59×)	12.34	(1.55×)
1.90	(2.32×)	0.83	(1.01×)
23.60	(1.10×)	22.30	(1.04×)
22.90	(1.74×)	13.24	(1.01×)
161.59	(5.68×)	88.63	(3.12×)
154.74	(4.69×)	83.89	(2.54×)
5.30	(3.61×)	2.66	(1.81×)
30.68	(4.45×)	15.56	(2.26×)
103.95	(3.36×)	41.79	(1.35×)
Avg.:	3.30×		1.66×

4(c) Blast versus CFLAT and OAT adapted to local logging.

Table 4: Performance of Blast on the Embench-IOT benchmark suite. Numbers reported are average of 10 runs, and the standard deviation is < 0.2%. Blast preallocated a log of size 1MB split into two 512KB symmetric halves with sentinel pages.

The performance of these benchmarks improves significantly with function inlining, which eliminates the instrumentation at the direct calls that are inlined. We configure the compiler to inline small functions (with fewer than 1000 LLVM IR instructions) up to a call-depth of 5. That is, at each direct call site, we check whether the called function has fewer than 1000 LLVM IR instructions; if so, we inline it. We recurse similarly for all direct calls in the inlined code, and repeat this at a call depth of 5. Such inlining improves the runtime performance of most benchmarks compared to the non-inlined counterpart, reducing the average runtime overhead across all benchmarks to 70%. The runtime overheads of `crc32`, `sglib-combined`, `st`, and `tarfind` reduces to 58%, 95%, 83%, and 83%, respectively, post-inlining. While inlining does increase the size of the binary by 162%, we note that this cost is offset by the fact that it results in many fewer log entries—effectively setting up a time-space tradeoff. Some benchmarks, such as `edn` perform worse with inlining, and hence we do not advocate blind use of function inlining. Considering the best of $\mathcal{P}$'s performance with and without inlining, we observe an average runtime overhead of 67% for whole-program path attestation across all our benchmarks.

Three factors contribute to the runtime overhead of Blast-instrumented binaries—Ball-Larus instrumentation, SFI instrumentation, and the log commit operation. Figure 8 shows how each of these factors contributes to the runtime overhead of the benchmarks. For this experiment, we show the breakdown of the overhead only for the inlined version of $\mathcal{P}$. Our results indicate that the log commit operation, which proceeds in parallel to $\mathcal{P}$, contributes negligibly to the overhead of most benchmarks, except in the case of `primecount` (the overhead manifests as waiting time in $\mathcal{P}$). The bulk of the overhead is due to Ball-Larus instrumentation, which is uniformly more than the overhead due to SFI instrumentation.

Performance with serial log commit. When an extra CPU core is not available for the log commit thread to execute, that thread must execute interleaved with $\mathcal{P}$. Because the CPU can either be in the normal world or in the TEE (secure world) at any given time, its state must be toggled based upon whether the thread performing the log commit operation is executing or whether $\mathcal{P}$'s thread is executing. Moreover, $\mathcal{P}$ may be required to wait if it has written a log half to capacity while the other half has not yet been committed. These factors contribute to overhead in the execution

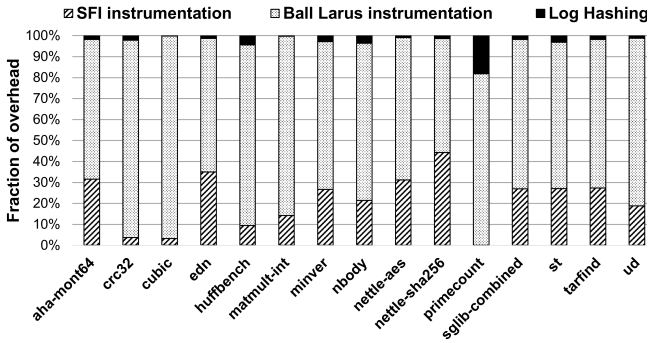


Figure 8: Contribution of SFI, Ball-Larus instrumentation, and log commit operations to overhead.

of $\mathcal{P}$. Table 4(b) reports the overhead observed with such serial execution on a single CPU core. For each benchmark, we report the lower of the runtime overheads observed for with either the inlined or non-inlined version of that benchmark. Observe that the average runtime overhead is 175%, compared to 67% when the log is committed in parallel. Some benchmarks, such as `crc32`, `matmul-int`, `primecount`, and `ud`, spend close to 50% of their total runtime waiting, showing the benefit of committing the log in parallel if an extra CPU core is available.

Comparison with CFLAT and OAT. As previously discussed, the number of domain switches in CFLAT and OAT imposes an unacceptably large overhead on the raw runtimes of the Embench-IOT programs. However, it is possible to adapt CFLAT and OAT to also perform local logging. Even in this setting, Blast's approach inserts fewer entries into the log compared to CFLAT and OAT. We modified CFLAT and OAT to perform local logging, and integrated them with SFI to protect `LogReg`, as done in Blast.

Table 4(c) shows the performance of the benchmarks on the local-logging versions of CFLAT and OAT. We assume the presence of a dedicated CPU core to execute the log commit thread, *i.e.,* the parallel log commit setting. Blast outperforms CFLAT and OAT by up to 5.74× and 3.12×, respectively, and on average, by 3.30× and 1.66×, respectively. The improvement is because Blast inserts fewer entries into the log as compared to CFLAT and OAT, thus triggering fewer log flushes to the TEE. However, note that the multiplicative factor by which Blast outperforms CFLAT or OAT

Embench-IOT Program ↓	Energy Consumption in Joules	
	Baseline	Blast (Overhead)
aha-mont64	34.37	88.59 (158.52%)
crc32	34.24	59.84 (74.75%)
cubic	5.24	6.03 (15.14%)
edn	76.19	146.42 (92.17%)
huffbench	40.57	82.93 (104.39%)
matmult-int	97.48	141.38 (45.03%)
minver	12.74	25.17 (97.57%)
nbody	1.50	2.53 (68.54%)
nettle-aes	48.28	64.61 (33.80%)
nettle-sha256	35.00	39.80 (13.71%)
primecount	44.92	91.23 (103.71%)
sglib-combined	49.29	104.77 (112.58%)
st	2.31	4.64 (101.01%)
tarfind	10.93	21.80 (99.43%)
ud	52.74	99.77 (89.16%)
Average overhead:		**80.60%**

Table 5: Energy consumption with and without Blast on Embench-IOT benchmarks. The input voltage to the Raspberry Pi3 is constant at 5.1V, and the overall energy consumption is the product of the voltage, current draw, and the execution time of the benchmark.

as reported in Table 4(c) is somewhat smaller than the multiplicative factor by which Blast reduces the number of log entries as compared to CFLAT and OAT (as reported in Table 3). This is because of the additional register operations to `BLReg` in Blast that are required by the Ball-Larus approach, which are absent in both CFLAT and OAT.

4.2 Evaluating Energy Consumption in Blast

Blast-instrumented programs have additional instructions to enable path measurement, logging, and periodic commitment of the log state to the TEE. We conducted an experiment to evaluate the energy overhead of executing these additional instructions on our Raspberry Pi3 Model B+ evaluation platform.

To conduct these experiments, we connected a Hioki 3274 current probe to the 5.1V DC power supply of the Raspberry Pi3 board. We paired this with a Tektronix TBS1064 four channel digital storage oscilloscope that runs at 60MHz, and has a sampling rate of 1GS/s. This setup measures the current (in mA) drawn by the Raspberry Pi3 board every 50ms. Multiplied with the constant input voltage of 5.1V, this yields the instantaneous power draw (in mW), which we integrated over the execution time of each benchmark to obtain the overall energy consumption.

We compared the energy consumption of unmodified Embench-IOT benchmarks with the best performing Blast-instrumented variant from Table 4(a). Table 5 reports the results of these experiments, which show an average increase in energy consumption of about 80% across the Embench-IOT benchmarks.

4.3 Effectiveness of the WPP Representation

We observe hundreds of millions of control-flow events when we execute Blast-instrumented versions of each benchmark (see Table 3 for raw log entry numbers). The size of the resulting log can therefore run into MBs (or even GBs), as shown in Table 6. Recall that Blast computes the cumulative hash over the log during the program execution and sends only a nonced, and digitally-signed 32-byte hash value to the verifier $\mathcal{V}$. However, as discussed in Section 3.4, $\mathcal{V}$ may request more information from the prover. In such cases, transmitting large raw log files is infeasible, and the prover resorts to computing and sending a WPP. Table 6 compares the raw size of the log, a version of the log compressed with the `bzip2` utility [13], and the WPP representation of the log. Bzip2 uses

Embench-IOT Program ↓	Raw log size (MB)	bzip2 file size (bytes)	WPP size (bytes)
aha-mont64	724.5MB	475,740 bytes	768 bytes
crc32	664.7MB	33,490 bytes	147 bytes
cubic	1.2MB	233 bytes	216 bytes
edn	1376.6MB	211,078 bytes	818 bytes
huffbench	889.8MB	4,706,860 bytes	9750 bytes
matmult-int	1477.7MB	105,882 bytes	370 bytes
minver	215.9MB	63,145 bytes	699 bytes
nbody	17.6MB	2,051 bytes	408 bytes
nettle-aes	195.2MB	40,022 bytes	843 bytes
nettle-sha256	132.3MB	35,055 bytes	336 bytes
primecount	1076.8MB	23,034,525 bytes	73,478 bytes
sglib-combined	910.0MB	421,6020 bytes	6,716 bytes
st	34.7MB	3,784 bytes	476 bytes
tarfind	184.6MB	382,229 bytes	257,756 bytes
ud	975.4MB	297,473 bytes	533 bytes

Table 6: Comparison of the size of the raw log against a log compressed with two lossless approaches: bzip2 and WPP.

Open Syringe Pump Bolus (ml) ↓	Baseline	Blast	
	Time (s)	Time (s)	Overhead
0.5 ml	1.28	1.42	(+10.93%)
1 ml	2.56	2.71	(+5.86%)
2 ml	5.12	5.28	(+3.13%)

Table 7: Performance evaluation for different bolus amounts. Bolus (*mL*) is the dose of drug.

a lossless compression technique based on the Burrows-Wheeler block sorting text compression algorithm and Huffman coding. The WPP construction algorithm identifies and compactly represents repeated fragments in structure of the log using a context-free grammar. Table 6 clearly shows the benefits of the WPP representation, which is just a few hundred *bytes* in each case.

4.4 Case Study: Open Syringe Pump Benchmark

We evaluated Blast using the Open Syringe pump benchmark [40], an implementation of a medical syringe pump for low-end embedded devices. We chose this benchmark because it has also been used by prior work to evaluate the security and performance of their systems. We used the publicly-available version of the application that the CFLAT authors ported for their work. The application takes a numeric input which sets the quantity of bolus (`set-quantity`), and a trigger input (+/-) which moves the syringe pump to dispense or withdraw the set bolus (`move-syringe`).

We conducted a performance evaluation of the instrumented Open Syringe pump benchmark similar to CFLAT by executing it with different bolus quantities. As shown in Table 7, the raw *whole-program* path attestation overhead incurred using the Blast-instrumented application is 0.14s for 0.5ml, 0.15s for 1ml, and 0.16s for 2ml. In contrast, the raw performance overhead of the corresponding CFLAT-instrumented benchmark is 1.2s for 0.5ml, 2.4s for 1ml, and 4.8s for 2ml for *attesting paths only in the* `set-quantity` *and* `move-syringe` *functions* [2, Section 6.3]. Further, we observed only a single TEE domain switch in the Blast-instrumented version of the application. This is because the number of control-flow events observed during the execution of this application did not fill the log to capacity, *i.e.,* there were fewer entries than to even fill one half of the space allocated for the log. In contrast, CFLAT makes a TEE domain switch on every basic block.

Next, we illustrate the effectiveness of WPP at detecting anomalous behaviour of the `set-quantity` function. The functionality of the application is such that depending on the bolus amount, the stepper motor runs a different number of steps, which is implemented as a loop in the ported application. The CFG consists of three acyclic paths: a loop entry path with Ball-Larus value 1, a

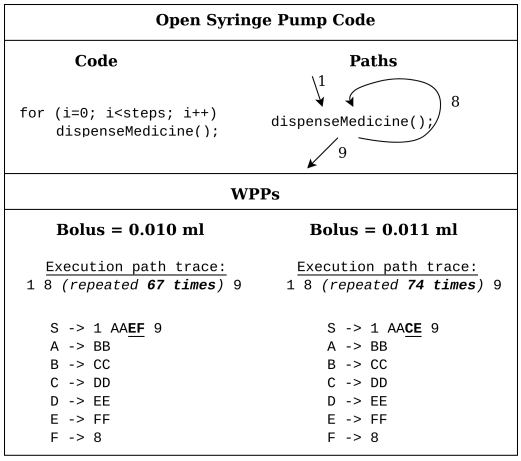


Figure 9: Code snippet of Open Syringe Pump showing the Ball Larus path numbers (we have omitted the function names for brevity), accompanied with execution path traces and WPPs for different bolus amounts, 10 μL and 11 μL.

loop iteration path with value 8, and a loop exit path with value 9 as shown in Figure 9. The loop runs 68 iterations for 10μL, and 75 iterations for 11μL, *i.e.,* path number 8 occurs 67 times in path trace for 10μL, and 74 times in path trace for 11μL. Both path traces start and end with path numbers 1 (loop entry) and 9 (loop exit). We consider an attack on the program that causes it to pump 11μL when it is expected to pump 10μL. The WPPs capture the differing number of loop iterations in the path traces and are easily distinguishable to a verifier $\mathcal{V}$ based on the number of bolus given as input. Blast similarly detects errant behavior in the `move-syringe` function since the trigger inputs (+/-) causes the program to take different paths in the CFG.

5 RELATED WORK

Remote attestation has been explored in various domains [4, 8, 24, 35, 47] to establish the absence of malicious changes to the memory content. Software-based attestation techniques such as SWATT [45] and PIONEER [44] were originally proposed for low-end embedded devices because TEEs were generally unavailable (historically) on low-end embedded devices. These methods implemented attestation via a self-checksumming function, implemented entirely in software. The security of these techniques relies on precise timing measurements, and is applicable only in settings where the communication delay between $\mathcal{V}$ and $\mathcal{P}$ is deterministic, *e.g.,* communication between peripheral and host CPU [35]. Moreover, attacks have been proposed on software based attestation [14, 51].

Hybrid attestation techniques provide the same security guarantees as hardware-based approaches while minimizing modifications to the underlying hardware platform and reducing the assumptions of software-based approaches. VRASED [37] implements integrity-ensuring functions (*e.g.,* MAC) in software and uses trusted hardware to control the execution of this function. These attestation techniques share a standard limitation—they measure the state of the prover only when it executes remote attestation. They do not provide information about the program before measurement or its state between two consecutive measurements, and thus suffer from time-of-check to time-of-use (TOCTTOU) attacks. RATA [38] proposes a hybrid design to avoid TOCTTOU in microcontroller units. Blast is complementary to these approaches and focuses on attesting the program's control flow.

DIAT [4] is an integrity measurement approach that, like Blast, attempts to address the issue of performance overheads of path attestation. DIAT decomposes the program $\mathcal{P}$ into modules and attests selective modules that process specific data of interest. DIAT ensures that the communication between the modules takes place over a well-defined interface that allows data-flow tracking between the modules. In DIAT the modules that are attested are identified beforehand, and $\mathcal{V}$ does not have flexibility in choosing which modules to attest. LAPE [29] also uses an approach similar to DIAT for firmware attestation by splitting firmware into modules. Blast provides whole-program attestation in contrast to DIAT. Methods from Blast can also be used to improve the attestation of the modules that DIAT identifies, thus resulting in a better system overall. LO-FAT [22], LiteHAX [21], Atrium [52] and Tiny-CFA [39] introduce hardware support to reduce the overheads of control-flow path attestation on the prover and securely store measurements.

ReCFA [53] proposes an alternative approach to optimizing path attestation. It proposes a multi-phase program analysis to reduce the number of control events to be recorded at runtime. ReCFA's call-site filtering analyzes the control-flow graph and elides recording a call to a function if it is guaranteed to follow a call to its predecessor. ReCFA's control-flow event folding compresses the amount of data recorded in the integrity measurement sent to the verifier. The verifier itself uses a form of abstract execution to check the integrity of $\mathcal{P}$'s execution. ReCFA uses Intel MPK [30] for lightweight protection of the integrity of critical data structures used for call-site filtering and control-flow event folding within $\mathcal{P}$'s address space. ReCFA's methods are complementary to Blast's—in particular, ReCFA can benefit from Blast's Ball-Larus-based instrumentation placement, and vice-versa. However, ReCFA did not evaluate the performance impact of domain switches into the TEE, which are a key part of the design of path attestation methods for embedded devices and also severely impact their performance. Also, ReCFA cannot be implemented securely on embedded systems, which lack Intel MPK support that is presently available only on Intel server-class chipsets. Although similar in design to Intel MPK, ARM Memory Domains [7, 18] require a trap to the kernel to switch memory domains unlike MPK, in which the corresponding operation is done entirely in user-space.

MG-CFA [28] proposes dual attestation at function-level: (1) fine-grained that records every branch, function call and return events, or (2) coarse-grained that records only function entrance and exit events. It associates a vulnerable probability p with every function (which is predicted using ML models). Functions with p above a threshold are called vulnerable functions and others are normal functions. Vulnerable functions are checked at fine-granularity, others are checked at coarse-granularity. They evaluate their work on Raspberry Pi with ARM Trustzone and reports upto 600× overhead on real-benchmarks when all functions are checked at fine-granularity. These numbers are consistent with the results that we observed when we attempted to scale CFLAT and OAT to whole-program path attestation (Table 2). Blast's overhead, in contrast, is an average of 1.67× on the Embench-IOT benchmark suite.

6 CONCLUSION

Whole program path attestation has proven to be an elusive goal. This paper showed that prior methods for path attestation fail to scale to whole program paths because of the prohibitive number of TEE domain transitions that they make. We then proposed Blast, which combines local logging, optimal instrumentation placement in $\mathcal{P}$ and software fault isolation in a novel way, thereby enabling whole-program path attestation with runtime performance overheads of 67% on a set of embedded benchmarks.

Acknowledgements. (1) Raghavan Komondoor and the CCS'23 reviewers for their insightful comments; (2) The Department of Science and Technology, the National Security Council of India, and a Prime Minister's Research Fellowship for financial support.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Sys. Sec.*, 13(1), 2009.
[2] T. Abera, N. Asokan, L. Davi, J. Ekberg, T. Nyman, A. Paverd, A-R. Sadeghi, and G. Tsudik. C-FLAT: Control-Flow attestation for embedded systems software. In *ACM Conference on Computer and Communications Security*, 2016.
[3] T. Abera, R. Bahmani, F. Brasser, A. Ibrahim, A-R. Sadeghi, and M. Schunter. DIAT: Data integrity attestation for resilient collaboration of autonomous systems. In *ISOC Network and Distributed Systems Security Symposium*, 2019.
[4] T. Abera, R. Bahmani, F. Brasser, A. Ibrahim, A-R. Sadeghi, and M. Schunter. DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems. In *Networked and Distributed Systems Security Symposium*, 2019.
[5] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy*, 1997.
[6] ARM security technology. https://developer.arm.com/documentation/PRD29-GENC-009492/c.
[7] Memory domains (page b3-31). ARM v7A/v7R Architecture Reference Manual.
[8] N. Asokan, F. Brasser, A. Ibrahim, A-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann. Seda: Scalable embedded device attestation. In *ACM Conference on Computer and Communications Security*, 2015.
[9] A. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world. In *ACM Conf. on Computer and Communications Security*, 2014.
[10] T. Ball and J. Larus. Efficient path profiling. In *29th Annual ACM/IEEE Symposium on Microarchitecture*, Dec 1996.
[11] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *6th ACM Symposium on Information, Computer and Communications Security*, 2011.
[12] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *ACM Conference on Computer and Communications Security*, 2008.
[13] bzip2. Linux man page. https://linux.die.net/man/1/bzip2.
[14] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the ACM conference on Computer and communications security*, 2009.
[15] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming Without Returns. In *ACM Conference on Computer and Communications Security*, 2010.
[16] L. Chen, R. Landfermann, H. Löhr, M. Rohe, A-R. Sadeghi, and C. Stüble. A protocol for property-based attestation. In *Proceedings of the first ACM workshop on Scalable trusted computing*, pages 7–16, 2006.
[17] L. Chen, H. Löhr, M. Manulis, and A-R. Sadeghi. Property-based attestation without a trusted third party. In *International Conference on Information Security*, pages 31–46. Springer, 2008.
[18] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. Shreds: Fine-grained execution units with private memory. In *IEEE Symposium on Security and Privacy*, 2016.
[19] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *IEEE Symposium on Security and Privacy*, 2017.
[20] T. Cloosters, D. Paaßen, J. Wang, O. Draissi, P. Jauernig, E. Stapf, L. Davi, and A-R. Sadeghi. RiscyROP: Automated Return-Oriented Programming Attacks on RISC-V and ARM64. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, 2022.
[21] G. Dessouky, T. Abera, A. Ibrahim, and A-R. Sadeghi. LiteHAX: lightweight hardware-assisted attestation of program execution. In *International Conference on Computer Aided Design*, 2018.
[22] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A-R. Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017.
[23] Embench^{TM}: A Modern Embedded Benchmark. https://www.embench.org/.
[24] T Fraser, J Molina, and W Arbaugh. Copilot–a coprocessor-based kernel runtime integrity monitor. In *USENIX*, 2004.
[25] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *12th National Computer Security Conference*, pages 305–319, 1989.
[26] X. Ge, H. Vijayakumar, and T. Jaeger. Sprobes: Enforcing Kernel Code Integrity on the TrustZone. In *IEEE Workshop on Mobile Security Technologies*, 2014.
[27] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *IEEE Symposium on Security and Privacy*, 2016.
[28] J. Hu, D. Huo, M. Wang, Y. Wang, Y. Zhang, and Y. Li. A probability prediction based mutable control-flow attestation scheme on embedded platforms. In *IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom/BigDataSE)*. IEEE, 2019.
[29] D. Huo, Y. Wang, C. Liu, M. Li, Y. Wang, and Z. Xu. LAPE: A Lightweight Attestation of Program Execution Scheme for BareMetal Systems. In *22nd IEEE Intl. Conference on High Performance Computing and Communications*, 2020.
[30] Intel. Intel-64 and IA-32 architectures software developer's manual, 2018. https://software.intel.com/en-us/articles/intel-sdm.
[31] Intel SGX for Linux. https://github.com/intel/linux-sgx.
[32] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 115–124. IEEE, 2009.
[33] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu. Securing real-time microcontroller systems through customized memory view switching. In *Networked and Distributed Systems Security Symposium*, 2018.
[34] J. Larus. Whole program paths. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation*, May 1999.
[35] Y. Li, J. M. McCune, and A. Perrig. Viper: Verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 3–16, 2011.
[36] A. J. Neto and I. Nunes. ISC-FLAT: On the Conflict Between Control Flow Attestation and Real-Time Operations. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2023.
[37] I. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik. Vrased: A verified hardware/software co-design for remote attestation. In *28th USENIX Security Symposium USENIX Security*, pages 1429–1446, 2019.
[38] I. Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik. On the TOCTOU problem in remote attestation. In *ACM Conf. on Comp. and Comm. Sec.*, 2021.
[39] I. Nunes, S. Jakkamsetti, and G. Tsudik. Tiny-CFA: Minimalistic control-flow attestation using verified proofs of execution. In *DATE Conference*, 2021.
[40] Open syringe pump. https://github.com/manimino/OpenSyringePump.
[41] Open portable trusted execution environment. https://www.op-tee.org/.
[42] A-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *Proceedings of the 2004 workshop on New security paradigms*, pages 67–77, 2004.
[43] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security*, 2004.
[44] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. Van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM Symposium on Operating Systems Principles*, 2005.
[45] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, 2004.
[46] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.
[47] Z. Sun, B. Feng, L. Lu, and S. Jha. OAT: Attesting operation integrity of embedded devices. In *IEEE Symposium on Security and Privacy*, 2020.
[48] F. Toffalini, E. Losiouk, A. Biondo, J. Zhou, and M. Conti. Scarr: Scalable runtime remote attestation for complex systems. In *International Symposium on Research in Attacks, Intrusions and Defenses*, 2019.
[49] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles*, 1993.
[50] M. Walfish and A. Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2), February 2015.
[51] G. Wurster, P. C. Van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 127–138. IEEE, 2005.
[52] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A-R. Sadeghi. ATRIUM: Runtime attestation resilient under memory attacks. In *International Conference on Computer Aided Design*, 2017.
[53] Y. Zhang, X. Liu, C. Sun, D. Zeng, G. Tan, X. Kan, and S. Ma. ReCFA: Resilient Control-Flow Attestation. In *Annual Computer Security Applications Conf.*, 2021.