Whole-Program and Non-Bare-Metal Control-Flow Attestation

A THESIS

SUBMITTED FOR THE DEGREE OF

Doctor of Philosophy

IN THE

Faculty of Engineering

BY

Nikita Yadav

Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012 (INDIA)

January 19, 2026

Declaration of Originality

I, **Nikita Yadav**, with SR No. **04-04-00-10-12-20-1-18606** hereby declare that the material presented in the thesis titled

Whole-Program and Non-Bare-Metal Control-Flow Attestation

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2020-2025**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Vinod Ganapathy Advisor Signature

DEDICATED TO

My Mentors

whose guidance and support have shaped my journey

Acknowledgements

I would like to express my deepest gratitude to my Ph.D. advisor, Prof. Vinod Ganapathy, for being the greatest source of inspiration and a steadfast pillar throughout my academic journey. I owe much of my growth and achievements to his unwavering support, trust in my abilities, and insightful guidance. His immense patience and eagerness to discuss and brainstorm research ideas made our interactions both intellectually stimulating and encouraging. I always felt safe going to him with half-baked ideas or even just confusion, he never rushed me and always took the time to talk things through.

Prof. Vinod's enthusiasm and encouragement energized me, even during the most challenging times. I am also really thankful for how much he supported my applications and encouraged me to follow my research pursuits. I greatly appreciated his advising style, he gave me the freedom to explore new directions while always helping me stay grounded and focused. I feel truly honored to have been his student, and I cannot thank him enough for his unwavering support.

I am also thankful to my collaborators, whose contributions have shaped much of my research. At IBM Research, I worked with Dushyant Behl and Praveen Jayachandran. Dushyant mentored me during my internship and remained a steady source of support long after. I learned a great deal from his clear and thoughtful problem-solving approach, and always appreciated his generosity and clarity. At Witness Chain, I had the pleasure of working with Prof. Himanshu Tyagi, Vishal Sevani, and Peyaio Sheng. Prof. Himanshu's passion for research and high energy were inspiring. Vishal is one of the most hardworking and patient person I have worked with, and I truly enjoyed our collaboration. I also appreciated working with Peyaio and the good memories we shared at the CCS conference. I am grateful to Prof. Ahmad-Reza Sadeghi for the opportunity to work with his group at TU Darmstadt—it was a truly enriching experience. I also extend my sincere thanks to Prof. Georgios Smaragdakis and Prof. Alexios Voulimeneas at TU Delft for their support and collaboration.

Warm thanks to my colleagues at the CSSL lab—Arun, Kripa, Hrushikesh, Kunal, Vivek, Eikansh, Isha, Dev, Gokul, Himanshu, Kanmani, Rakesh, Rounak, and Shivraj—for creating a cheerful and supportive environment. I collaborated with Arun on multiple projects and admired his calm, composed approach even under pressure. Kripa was always eager to help, especially with debugging; his energy made the lab a more vibrant place. Hrushikesh was a great collaborator, and I also appreciated working with Kunal and Dev.

I would like to thank Ms. Padmavathi, Ms. Kushael, and the other staff at the CSA office for their kind assistance on the administrative front. I gratefully acknowledge financial support from the Prime Minister's Research Fellowship, the Google PhD Fellowship, the CROSSING Fellowship at TU Darmstadt and the Murthy-Govindaraju Research Endowment Award, established by IISc alumnus Dr. Madhukar Govindaraju and Dr. Chaya Murthy. This generous financial assistance helped make my Ph.D. journey smoother and largely worry-free.

I have truly cherished my time at IISc, and one of the biggest reasons has been its beautiful, green campus. The serene environment never ceased to captivate me and brought a sense of peace and grounding that I will deeply miss. The excellent campus facilities including the mess, health center, and gymkhana, played a crucial role in helping me stay healthy and focused, allowing me to devote most of my time to my research without worrying about day-to-day needs. It has truly been a blissful home, the place I have enjoyed the most in my entire life. I tried to capture my feelings for IISc in a short poem, with some help from ChatGPT:

Like a wanderer lost in an endless wood,
In awe, I stand, feeling blessed under your hood.
Though years have passed, I still can't comprehend
Will this feeling fade, or stay with me till the end!

Last but not least, I thank my friends at IISc for standing by me in difficult times and making this journey joyful. I am especially grateful to Rohit, Nabanita, Sukeerti, and Srishty for their unwavering friendship and support. A special thanks to Marco Chilese and Franziska Vollmer at TU Darmstadt for helping me settle in and making my time there memorable.

Finally, I thank my family for believing in my decision to pursue a Ph.D. I am especially grateful to my mother for her endless support, and to my brothers, Chetan and Ketan, for standing by me in every phase. Special thanks to my sister-in-laws, Naina and Vandana, for being more like sisters and encouraging me through tough times.

Thank you!

Abstract

Software attestation enables a remote entity to verify that a program executed as intended on an untrusted platform, with applications in domains such as embedded systems, Internet of Things (IoT), and cloud-edge computing. Control-Flow Attestation (CFA) is one such approach that verifies a program's control-flow integrity by recording the path taken during its execution. In CFA, a prover platform convinces a remote verifier of the integrity of the prover program by recording the path that the program takes as it executes a particular input. While a number of techniques have been developed for CFA, there are two critical issues that have not been addressed. First, prior techniques have generally been applied to record paths only for parts of the prover program's execution and does not scale to attestation of whole programs. Second, they are designed for bare-metal environments and are therefore not applicable in non-bare-metal environments. CFA measurements must be collected and stored securely, *e.g.,* within a Trusted Execution Environment(TEE) on the platform where the program executes, failing which it will not be an accurate record of the program's execution. These requirements are satisfied relatively easily when the program executes atop bare-metal hardware, as most prior CFA approaches have assumed.

This dissertation develops new tools and techniques to address these two key challenges of the prior CFA techniques. It proposes efficient path measurement strategy and demonstrates that CFA can be applied securely to attest userspace programs running in non-bare-metal environments.

Specifically, this dissertation makes the following contributions. First, it considers the problem of *whole program control-flow attestation*, *i.e.,* to attest the execution of the entire prover program. It shows that prior approaches for CFA use sub-optimal techniques that fundamentally fail to scale to whole program paths, and impose a large runtime overhead on the execution of the prover program. It then develops Blast, an approach that reduces these overheads using a number of novel techniques inspired by prior work from the program profiling literature. Experimental results show that Blast enables whole-program CFA with an average runtime overhead of 67% on a suite of embedded benchmarks, making it practical for a wide range of applications.

Second, it considers *non-bare-metal control-flow attestation*, in which the prover program executes as a user-level process atop an operating system (OS) on the remote computing platform. It shows that prior approaches to CFA are insecure in the presence of OS-level adversaries. It describes the design and implementation of a system called SULFUR that securely enables CFA of user-space applications in non-bare-metal settings. SULFUR accomplishes this goal via a co-developed OS called SULFUROS. SULFUROS makes novel use of security extensions available in commodity AArch64 hardware—Privileged-Access Never (PAN) and Privileged-Execute Never (PXN). It experimentally shows that SULFUR enables secure CFA with low additional runtime overheads in non-bare-metal environments. Specifically, SULFUR contributes only an additional 1.94% overhead for CFA-based attestation of a program, compared to CFA-based attestation atop a commodity Linux OS.

Together, these contributions advance the state-of-the-art in CFA by enabling scalable whole-program attestation and extending its applicability to non-bare-metal environments.

Publications based on this Thesis

1. **Nikita Yadav**, and Vinod Ganapathy. *Whole-Program Control-Flow Path Attestation*. In Proceedings of CCS'23, the 30th ACM SIGSAC Conference on Computer and Communications Security, Copenhagen, Denmark, November 2023, ACM Press. [95]

2. **Nikita Yadav**, Hrushikesh Salunke, Dev Tejas Gandhi, and Vinod Ganapathy. *Non-Bare-Metal User-Space Control-Flow Attestation*, In Proceedings of ACSAC'25, the 41st Annual Computer Security Applications Conference, Honolulu, Hawaii, USA, December. 2025 [97]

Publications outside of this Thesis

1. Arun Joseph, **Nikita Yadav**, Vinod Ganapathy, Dushyant Behl, and Praveen Jayachandran. *Data Protection in Permissioned Blockchains using Privilege Separation.* In Proceedings of COMSNETS'23, the 15th International Conference on Communication Systems and Networks, Bangalore, India, January 2023. [60]

2. Arun Joseph, **Nikita Yadav**, Vinod Ganapathy, and Dushyant Behl. *A Contributory Public-Event Recording and Querying System.* In Proceedings of SEC'23, the 8th ACM/IEEE Symposium on Edge Computing, Wilmington, Delaware, December 2023. [59]

3. Peiyao Sheng, **Nikita Yadav**, Vishal Sevani, Arun Babu, SVR Anand, Himanshu Tyagi, Pramod Vishwanath. *Proof of Backhaul: Trustfree Measurement of Broadband Bandwidth.* In Proceedings of NDSS'24, the 31st Annual Network and Distributed System Security Symposium, San Diego, California, February/March 2024, Internet Society. [85]

4. **Nikita Yadav**, Franziska Vollmer, Ahmad-Reza Sadeghi, Georgios Smaragdakis, Alexios Voulimeneas. *Orbital Shield: Rethinking Satellite Security in the Commercial Off-the-Shelf Era.* In Proceedings of 3S'24, the 1st Security for Space Systems Conference, Noordwijk, Netherlands, May 2024. [96]

5. David Koisser, Richard Mitev, **Nikita Yadav**, Franziska Vollmer, Ahmad-Reza Sadeghi. *Orbital Trust and Privacy: SoK on PKI and Location Privacy Challenges in Space Networks.* In Proceedings of the 33rd USENIX SECURITY Symposium, Philadelphia, PA, USA, August 2024. [65]

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Our focus in this dissertation is on *control-flow path attestation*, or simply, *control-flow attestation (CFA)*. This dissertation presents techniques to strengthen CFA by scaling it to attest entire program executions and by making it robust for deployment in non-bare-metal environments.

1.1 Motivation and CFA

Modern computing systems are increasingly distributed, spanning devices across the cloud, edge, and Internet of Things (IoT). In such environments, software often executes on remote or untrusted platforms that may be compromised or tampered with. Ensuring that remote software executes as intended (without unauthorized modifications) has therefore become a critical challenge in building trustworthy systems.

Attestation allows a remote *verifier* to establish trust in the integrity of a remote 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) [79] on early systems, or other hardware-based trusted execution environments (TEE) such as the ARM TrustZone [18] or Intel SGX [56] 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 [9, 48], that the system satisfies certain properties [35, 36, 63, 78], or executed certain operations [87].

Control-flow attestation (CFA) is a security technology that enables fine-grained verification

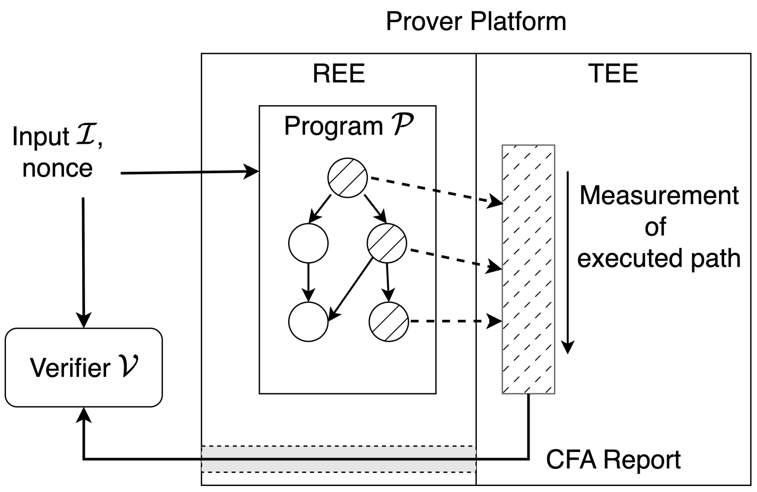


Figure 1.1: Basic setup for CFA. Instrumentation inserted in $\mathcal{P}$, executing in the REE, stores the CFA measurements in the TEE. The shaded region in the REE indicates that the CFA report is encrypted within the TEE, forming a secure channel through which it is transmitted via the REE to the verifier $\mathcal{V}$.

of the execution of a program running on a computing platform. In CFA, an input $\mathcal{I}$ is given to a program $\mathcal{P}$ running on a (remote) computing platform, which is expected to be equipped with a TEE, such as ARM TrustZone [10, 18] or similar. In response to input $\mathcal{I}$, the computing platform records the execution path (or relevant portions thereof) of the program $\mathcal{P}$ as it executes $\mathcal{I}$, while interfacing with the TEE to store the recorded information (called a *measurement)* securely. The computing platform then sends the measurement recorded in the TEE (together with the input $\mathcal{I}$) to a *verifier* $\mathcal{V}$, typically in response to a challenge from $\mathcal{V}$—this information is digitally signed by the TEE. $\mathcal{V}$ can then use the recorded measurement to check that $\mathcal{P}$ indeed executed as expected on input $\mathcal{I}$. The computing platform and $\mathcal{P}$ are collectively called the *prover*, which must offer proof of execution of input $\mathcal{I}$ on $\mathcal{P}$.

Figure 1.1 shows the basic setup for CFA. CFA monitors the functionality of a program $\mathcal{P}$ as it executes on a remote computing platform, and processes inputs, *e.g.,* $\mathcal{I}$ as shown in the figure. $\mathcal{P}$ typically executes within the rich execution environment (REE) of that platform. For example, on an ARM TrustZone-based compute platform, $\mathcal{P}$ executes within the normal world/REE. Before $\mathcal{P}$ starts executing, the TEE performs static binary attestation of $\mathcal{P}$ that allows the verifier $\mathcal{V}$ to ensure that the binary has not been replaced.

The CFA process consists of the following two phases:

- **CFA Measurement, where control-flow measurements are collected.** $\mathcal{P}$ is instrumented so that it records the control-flow path taken by the program. As $\mathcal{P}$ executes on $\mathcal{I}$, the instrumentation in it tracks the path followed and stores this information—the CFA measurement—in the TEE. Prior works on CFA have proposed several instrumentation strategies as discussed in Section 1.3.

- **CFA Verification, where the measurements are checked by the verifier.** Upon request, the prover platform sends the recorded CFA measurements to the verifier $\mathcal{V}$, which can use it for verification and audit. 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 $\mathcal{P}$. Most prior methods for CFA require $\mathcal{V}$ to have access to a *measurement database*, which consists of the attestation measurements (*e.g.,* hashes, branch history) obtained by running $\mathcal{P}$ on several inputs in a benign environment. $\mathcal{V}$ simply matches the measurement obtained from the prover platform against this database to determine whether the path taken corresponds to the input $\mathcal{I}$ that was provided to the prover. Recent advances in CFA have improved upon this CFA measurement database to instead use more advanced data structures (*e.g.,* [38], Chapter 2).

The main advantage of CFA is that it provides the verifier with a non-repudiable proof of execution of the prover program $\mathcal{P}$ on the input $\mathcal{I}$. 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, CFA can be used to assure a verifier that a syringe pump embedded within a patient has indeed delivered the desired dosage of a drug [5], or that a robot arm has indeed completed a task in response to a command from a remote operator [87]. Because CFA 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. CFA can also be seen as a lighter-weight alternative to verifiable computation [91]. Deviations from the expected measurement either indicate that the input was corrupted during transmission, or that there was a security compromise or malfunction in the prover platform (*i.e.,* robot or embedded device), *e.g.,* due to a control-flow hijacking attack in $\mathcal{P}$ running on the platform. The measurement obtained by $\mathcal{V}$ can also be used as an audit trail to verify claims, *e.g.,* insurance or legal claims, made by the computing platform or the operator.

1.2 Threat Model and Assumptions of CFA

This section discusses the threat model and assumptions commonly made by CFA approaches. All the approaches discussed in this thesis rely on instrumenting $\mathcal{P}$ to collect CFA measurements at runtime. 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 $\mathcal{P}$.

CFA approaches assume that the prover platform has implemented standard data-execution prevention (DEP) 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 in $\mathcal{P}$ is key to precisely computing the CFA measurement that the verifier $\mathcal{V}$ will check.

Code-reuse attacks (*e.g.,* return-oriented [83] or jump-oriented programming [29, 34]), and attacks that maliciously modify program data [53] that alter the program path followed in $\mathcal{P}$ can readily be detected because the CFA measurement that is collected by the TEE on the prover will diverge from the value that $\mathcal{V}$ expects. Our focus in this thesis 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 [87]), but such instrumentation is orthogonal to the discussion in this thesis. The specific threat models for the two works (Chapter 2 and Chapter 3) presented in this thesis differ slightly; their precise assumptions and adversary capabilities are detailed separately in their respective chapters.

1.3 Prior Art in CFA

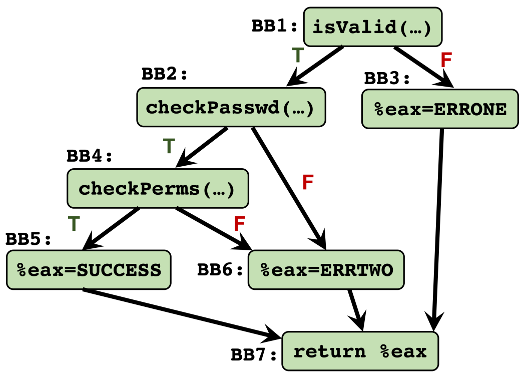
Prior work on CFA (*e.g.,* [5, 6, 44, 55, 74, 87, 89, 101]) has largely focused on programs that run atop embedded devices. Most of these approaches work by instrumenting the prover program so that at runtime, the instrumentation collects measurements that are committed to the TEE, *e.g.,* the secure world of an ARM TrustZone-enabled chip on the embedded device (Section 2.2.1). 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 the program input $\mathcal{I}$.

We now describe prior work on CFA with a focus on CFLAT [5] and OAT [87] as representative approaches. Other approaches [6, 44, 55, 74, 89, 101] use largely similar methods. We use the code snippet in Figure 1.2(a) as a running example denoting $\mathcal{P}$. Figure 1.2(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.

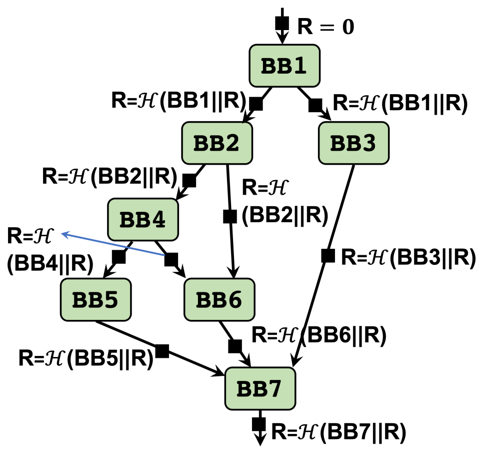
CFLAT instruments the CFG of each function to store a rolling hash of the code of the

```
if (!isValid(userID)) {
   return ERRONE;
}
if (checkPasswd(userID, passwd)) {
   if (checkPerms(userID)) {
      do_authorized_computation();
      return SUCCESS;
   }
}
return ERRTWO;
```

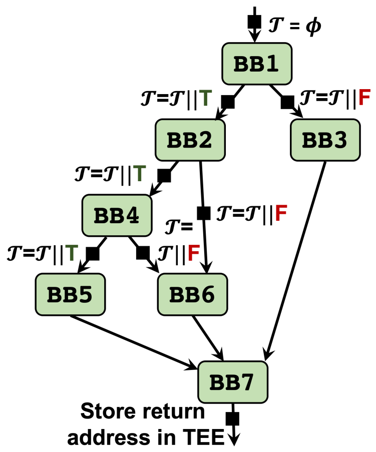
1.2(a) Code snippet of program $\mathcal{P}$.



1.2(b) CFG of Figure 1.2(a).



1.2(c) Instrumented by CFLAT.



1.2(d) Instrumented by OAT.

Figure 1.2: (a) Code snippet of running example program $\mathcal{P}$, (b) its corresponding control-flow graph (CFG), (c) CFG instrumented using CFLAT [5], and (d) CFG instrumented using OAT [87]. CFG edges with the black squares have instrumentation that domain switch into the TEE.

basic block IDs encountered during the execution of the function. CFLAT's CFA 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 1.2(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 1.2(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. Recall that $\mathcal{P}$ runs in the REE. CFLAT maintains the CFA measurement in the TEE, and each instrumentation contains a trampoline that performs a domain switch from REE 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 ID into the CFA 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 ID 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 CFA 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, 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 CFA measurements on a per-function basis. A verifier presented with CFLAT's CFA 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 CFA 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 CFA measurement for each operation. The CFA measurement consists of: (a) a bit trace

CFG edge	CFLAT	OAT
Conditional branch	✓	✓
Unconditional branch	✓	✗
Direct function call	✓	✗
Indirect function call	✓	✓
Return/function exit	✓	✓

Table 1.1: Control-flow edges in CFLAT and OAT that contain instrumentation to perform a domain switch to the TEE.

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 CFA measurement is stored in the TEE and updated via the instrumentation that OAT inserts in $\mathcal{P}$.

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 1.2(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 CFA measurements on a per-operation basis. The verifier $\mathcal{V}$ uses the direction of each conditional branch, and address of each indirect call or jump in CFA report, 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". Table 1.1 summarizes the CFG locations instrumented by CFLAT and OAT, each of which involves a domain switch from REE to the TEE.

1.4 Gaps in Prior Art and Our Advances

CFA has emerged as a powerful technique for verifying the integrity of program execution on remote devices. However, despite significant progress in this area, prior CFA approaches face two key limitations that restrict their applicability in real-world systems. First, existing approaches attest control-flow paths within specific regions of a program, such as annotated "operations," they fail to scale attestation for the program's entire execution. Partial path attestation leaves programs vulnerable to attacks targeting unmeasured regions, and we develop techniques to overcome this limitation and scale CFA to whole program attestation. Second, prior work predominantly assumes that the prover program runs directly on bare-metal hardware, an assumption that does not hold for higher-end embedded systems, where an operating system (OS) is typically present. This thesis shows that trusting the OS, as prior non-bare-metal CFA approaches have done, introduces critical vulnerabilities. We propose techniques to secure CFA even when programs execute atop potentially compromised operating systems.

The following subsections delve into these two advances in detail.

1.4.1 Scaling CFA to Whole Program Attestation

Unfortunately, as we show in this thesis, 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 $\mathcal{P}$. For example, OAT is tailored to attest the integrity of "operations" in embedded programs, which are specific regions of the program (manually 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 [54, 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.,* [5, 44, 74, 89]) 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. Thus, we aim to develop techniques to scale CFA to attest entire execution of a program.

We develop novel techniques to reduce runtime overhead of CFA and propose BLAST (Chapter 2). BLAST adapts a classic technique from the program profiling literature, called Ball-Larus profiling, to ensure optimal placement of the instrumentation in the program $\mathcal{P}$ that runs on the embedded device. We observed that the main reason for the high overheads incurred by prior methods was frequent invocations of the TEE. BLAST develops a novel method that used a combination of local logging and software-fault isolation to reduce this overhead. Further, BLAST adapts another classic idea called Whole-Program Paths from the program profiling literature to compactify the data gathered on the embedded device and reported to the verifier. In all, these methods brought down the overheads of CFA to an acceptable value of 67% from several orders of magnitude compared to prior works.

1.4.2 Enhancing Applicability of CFA

Significantly, prior works have largely targeted the *bare-metal setting*, *i.e.,* the setting in which $\mathcal{P}$ executes directly atop the bare-metal hardware of the underlying computing platform. These works consider TEE as the trusted computing base (TCB), and the prover program $\mathcal{P}$ as running atop bare-metal hardware in the untrusted REE. While the assumption of bare-metal execution may hold for specialized embedded devices, it does not hold for higher-end devices. For example, many embedded devices, robots and even satellites [45] contain powerful processors and allow concurrent execution of multiple applications. The presence of an REE OS greatly eases application development and isolation by enabling abstraction and virtualization of the hardware.

Although some projects [7, 89, 101] have considered CFA in the *non-bare-metal setting* in which $\mathcal{P}$ runs atop an operating system (OS) in the REE, they have generally trusted the REE OS. Their primary focus has been to scale CFA to attest complex user-space applications. Blindly trusting this REE OS risks bringing a large, potentially vulnerable OS code base into the TCB [24, 62].

As we show in this thesis, in non-bare-metal settings, the security assumptions made by prior CFA methods do not hold. Adversaries with control over the REE OS can completely subvert the assumptions required for CFA to operate securely. These attacks greatly limit the scenarios where CFA can be securely used.

To address these challenges, this thesis presents SULFUR, a system that enables robust CFA in non-bare-metal settings. SULFUR introduces a co-designed architecture comprising a trusted monitor and a modified operating system (SULFUROS) that leverage commodity hardware features available on AArch64 platforms. Specifically, SULFUR uses hardware feature, ARM privilege-access never (PAN), to protect CFA logs from a compromised OS, ensuring that the measurements remain untampered. It also integrates kernel-level instrumentation to protect the assumptions required for CFA to operate securely, *e.g.,* SULFUR ensures that DEP is never disabled, and the instrumentation in $\mathcal{P}$ remains untampered. Further, SULFUR protects the runtime state of $\mathcal{P}$ and the system by mediating all updates through the trusted monitor, which verifies each update before it is applied. By combining OS instrumentation, and hardware isolation, SULFUR secures CFA in non-bare-metal environments and supports CFA of applications with low performance overhead.

1.5 Problem Statement, Thesis Statement, and Details of Technical Contributions

Problem Statement: Prior CFA techniques incur high runtime overheads on whole-program attestation and are not secure in non-bare-metal settings. This thesis aims to develop whole-program and non-bare-metal control-flow attestation techniques.

Thesis Statement: We can scale CFA to whole programs and apply it in non-bare-metal settings by the novel use of hardware and software-based isolation techniques.

This dissertation supports the above problem and thesis statements and makes the following contributions:

- We demonstrate that the prior CFA approaches incur prohibitive runtime overheads and performed a detailed analysis to identify the causes of the runtime overheads (Chapter 2).
- We present, BLAST (Chapter 2), an approach to scale attestation to whole program paths. With the novel use of path profiling algorithm—*Ball Larus Profiling*—for CFA and local logging of CFA measurements, BLAST achieves an average runtime overhead of 67% for whole-program path attestation on embedded benchmarks (Embench-IOT), outperforming prior approaches that incur up to 100×-1000× slowdowns.
- We incorporate another idea from program-profiling literature—*Whole-Program Paths*—to offer a compact yet expressive representation of the recorded path measurement (Chapter 2). This reduces the CFA measurement log size from few hundred MBs to only few hundred bytes.

- We show that prior methods for CFA, which have focused on bare-metal settings, are insecure when applied to programs that run in *Non-Bare-Metal* settings (Chapter 3).
- We present the design and implementation of SULFUR (Chapter 3), a system that securely enables user-space CFA in non-bare-metal settings. SULFUR allows robust CFA measurement of a user-space program $\mathcal{P}$ that runs on top of a customized OS—SULFUROS—that leverages hardware support (PAN and PXN) to protect CFA state before it is committed to the TEE.
- Our evaluation of SULFUR (Chapter 3) with various benchmarks shows that it accomplishes its goals without excessive runtime performance overheads.

Chapter 2

Whole-Program Control-Flow Attestation

2.1 Motivation

When we applied prior approaches to attest whole program paths, *i.e.,* to attest the entire control-flow path taken by the program, in a suite of embedded benchmarks, we found that the approaches impose a prohibitive overhead on the runtime of the program (see Section 2.2). To mitigate this, we explored a sampling-based selective attestation approach (see Section 2.5), where only a fraction of function calls are instrumented for CFA. However, even at low sampling rates (e.g., 1% of function calls), benchmarks exhibited extreme slowdowns, up to 12,000%. These findings revealed that existing CFA designs fundamentally do not scale to whole-program attestation. We further 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 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.

This chapter presents **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 CFA measurements (Section 2.3). 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) [90]. 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 CFA 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* [26] 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 with loops; details in Section 2.3). 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 [66] 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 [1]), Blast is able to attest

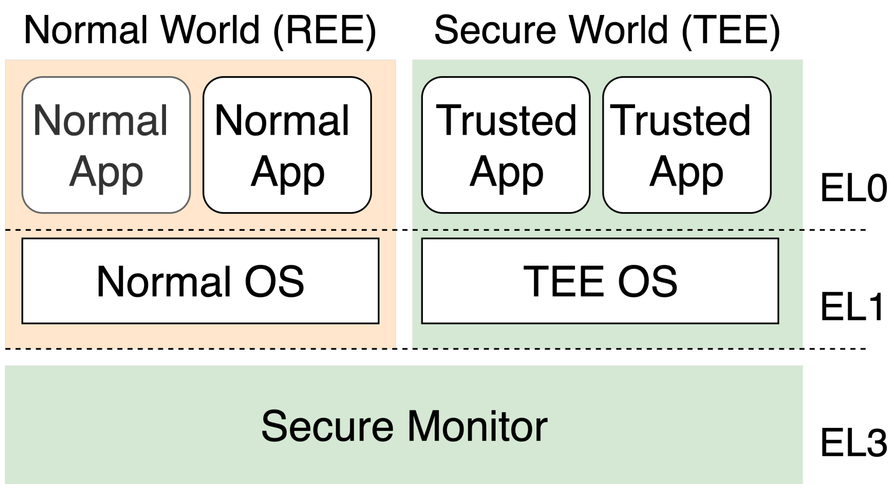


Figure 2.1: Overview of ARM TrustZone architecture. TrustZone divides the system into a secure world (TEE) and a non-secure/normal world (REE), with isolation enforced by hardware. The secure monitor (running at highest privilege level, Exception Level 3) manages transitions between the two worlds.

whole-program control-flow paths while imposing an average runtime overhead of 67%. This is significantly smaller than the overheads of competing CFA approaches when applied to whole program paths, which can slow the execution of the program by up to 100×-1000×.

2.2 Background

2.2.1 ARM Trustzone

This section introduces ARM Trustzone. The CFA techniques in this thesis rely on the use of ARM TrustZone as the TEE for CFA. ARM TrustZone is a widely deployed security architecture available in modern embedded devices. It provides a hardware-enforced mechanism to separate the system into two distinct environments as shown in Figure 2.1: the secure world or TEE and normal world or REE. The TEE hosts security-critical applications and data atop a trusted operating system, while the REE runs the general-purpose operating system and user applications.

TrustZone ensures strong isolation between these two environments. Specifically, the REE cannot access hardware resources, memory regions, software, interrupts, or peripherals assigned to the TEE. This separation is enforced by the hardware's security extensions, which label each transaction or access request as secure or non-secure, ensuring that only the TEE can access its designated resources.

The transitions between the TEE and REE are managed by the secure monitor, which executes at the highest privileged level. The secure monitor handles context switching between the two environments while preserving the integrity and confidentiality of the secure world.

This thesis leverages two key capabilities of ARM TrustZone. First, Trustzone provides secure storage that cannot be accessed by any software running in the REE, ensuring confidentiality and integrity of critical data. Second, TrustZone enables the generation of digital signatures using private keys securely stored within the TEE, providing strong authentication guarantees for measurements and attestation results.

2.2.2 Threat Model of BLAST

Blast relies on a compiler pass to insert the path measurement 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. Similar to prior works, Blast assume that the prover platform is equipped with a TEE, which the verifier $\mathcal{V}$ trusts. The prover program $\mathcal{P}$ can be hijacked by an adversary, *e.g.,* by feeding malicious inputs that exploit zero-day vulnerabilities in $\mathcal{P}$.

Further, prover platform has implemented standard data-execution prevention techniques to prevent code injection attacks in $\mathcal{P}$. Commodity operating systems generally provide data-execution prevention in concert with the hardware (*e.g.,* W$\oplus$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 [39, 64]. Such work is orthogonal to the focus of this work, and as in prior work [5, 87], 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 [76] to ease communication with the TEE, and OPTEE requires an operating system in the normal world. As we outline in Section 2.3.6, 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

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

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

Table 2.1: 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 ✓$_{\text{LOG}}$), Blast's instrumentation stores the path measurement in a local log and does not trigger a domain switch (see Section 2.3).

assume it is benign in intent.

Code-reuse attacks (*e.g.,* return-oriented [83] or jump-oriented programming [29, 34]), and attacks that maliciously modify program data [53] 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 work 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 work.

2.2.3 TEE Domain Switches in CFA

As must be clear from the description in Section 1.3, CFA requires extensive instrumentation to be inserted into the program $\mathcal{P}$. Table 2.1 summarizes the CFG locations instrumented by prior works—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 2.3, Blast's approach *does not* trigger a domain switch at these instrumentation locations. 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 CFA 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%.

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-

Embench-IOT	**Number of Control-Flow Events (by CFG edge types)**					**Total TEE Domain Switches**	
Program ↓	Conditional Branches	Unconditional Branches	Loop Headers	Direct Calls	Returns and Exits	Encountered at Runtime **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**

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 2.1. This figure also shows the number of branches that are loop header edges, which are the only branches at which BLAST inserts log entries.

Table 2.2: Number of TEE domain switches seen at runtime with CFLAT's and OAT's instrumentation for the Embench-IOT benchmark suite.

11.0.0, and instrumented benchmarks from the Embench-IOT suite [1]. Embench-IOT is a set of benchmarks that represents the requirements of modern connected embedded systems.

Table 2.2 reports the results of whole program CFA 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 2.3.

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 2.1), 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 2.2 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 ([87, Table II]); CFLAT's evaluation also suggests an overhead of $\sim$40μsec on their Raspberry Pi 2 evaluation platform ([5, Figure 8]). We used a Raspberry Pi 3 Model B+, for our evaluation of Blast, and observed an average world switch time of 190μsec on our platform. Observe that with such overheads for TEE domain switches and the numbers reported in Table 2.2, the raw runtimes for whole-program CFA of the benchmarks in Embench-IOT will run into several hours. Given that the baseline execution time of these benchmarks is just a few seconds (under 35 seconds in all cases, *cf.*, Table 2.4), whole-program CFA would impose *more than a 1000$\times$ 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$\times$ (Section 2.4.1).

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 CFA *do not scale to whole program CFA*, thereby motivating us to develop BLAST.

2.3 Design and Implementation of BLAST

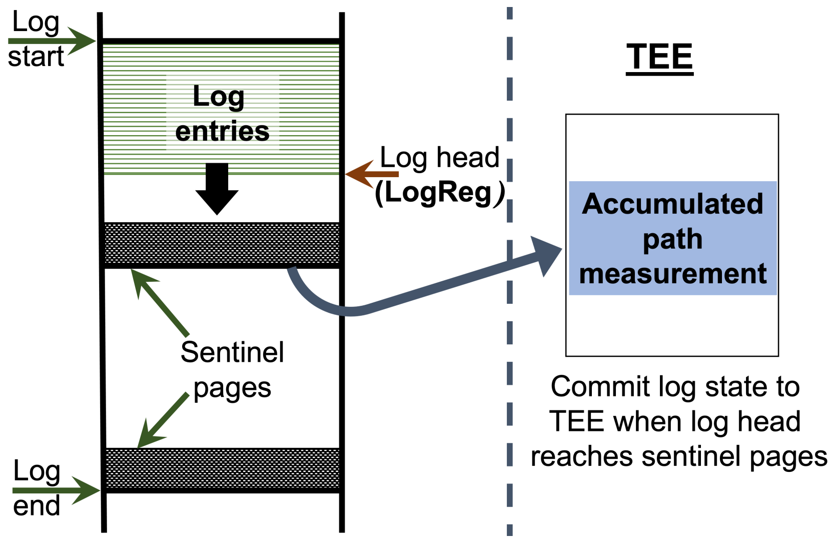
The analysis reported in the prior section shows that the key bottleneck that prevents prior methods from scaling to whole-program CFA is the number of TEE domain transitions that they perform. BLAST aims to reduce the number of TEE domain transitions encountered during CFA by performing *local logging* of control-flow events (Section 2.3.1). 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 [26], 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 domain transitions required to commit the log to the TEE when it reaches capacity (Section 2.3.2).

- *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 2.3.3).

BLAST processes the log to present a commitment of the whole-program control-flow path followed by $\mathcal{P}$ to $\mathcal{V}$ (Section 2.3.4). It creates a compact grammar-based representation of the log [66], 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 2.3.6).



Log start
Log entries
Log head (LogReg)
TEE
Accumulated path measurement
Sentinel pages
Commit log state to TEE when log head reaches sentinel pages
Log end

Figure 2.2: Structure of the log in Blast.

2.3.1 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 2.2 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 2.3.3).

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 2.4).

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.

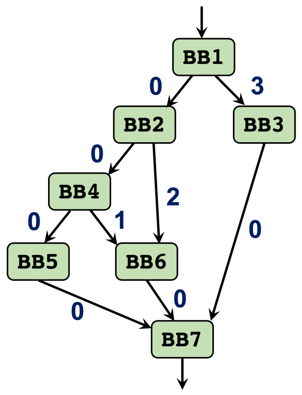
Blast's instrumentation to insert entries into the log simply adds the corresponding entry into the log and increments the value of `LogReg`. In particular, Blast does not perform any range checks on `LogReg` during each write, which would impose additional overhead on each store to the log. Instead, 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 2.3.4.

2.3.2 Ball-Larus Numbering for Optimal Logging

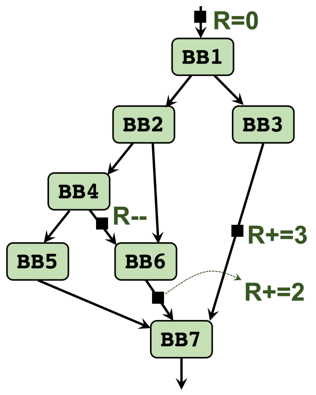
Blast builds upon the influential idea of Ball-Larus numbering [26]. The Ball-Larus algorithm places instrumentation in an optimal fashion by selectively adding instrumentation to the edges of a control-flow graph that increments the value of a counter in specific ways along specific edges. The instrumentation 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 $2\times$ more compact than instrumentation for bit-tracing (*á la* OAT) [26, 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.

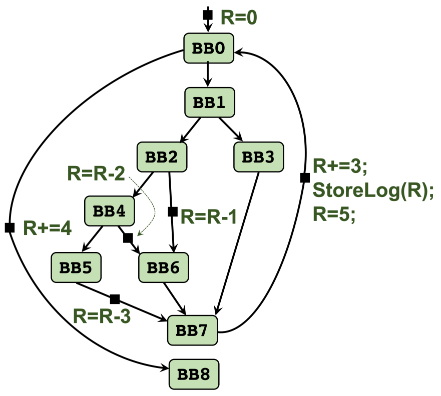
Given an acyclic CFG with uniquely-designated entry and exit nodes, the number of paths in the CFG is finite, say N. Ball-Larus numbering assigns a numeric value to each edge ($\mathsf{valEdge}(.)$)



2.3(a) Ball-Larus numbering of CFG in Figure 1.2(b).



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



2.3(c) Instrumented CFG from Figure 1.2(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

2.3(d) Path numbers for the CFG in Figure 2.3(c) obtained from the associated instrumentation.

Figure 2.3: Ball-Larus numbering and corresponding instrumentation. We also show how Ball-Larus numbering handles loops.

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 $\mathsf{valEdge}(.)$ 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 $\mathsf{valNode}(p)$ that denotes the number of paths from p to the exit node. At each step, it assigns values $\mathsf{valEdge}(e)$ to each outgoing edge e from p so that the following invariant is maintained—the sum of the $\mathsf{valEdge}(.)$ values of the edges from p to the exit node is a unique integer $\in [0, \mathsf{valNode}(p)-1]$. Algorithm 1, adapted from the original Ball-Larus paper [26], shows how valNode and valEdge are calculated for an acyclic CFG.

We are interested in the $\mathsf{valEdge}(.)$ values assigned to the edges of the CFG. Figure 2.3(a) shows one possible assignment of $\mathsf{valEdge}(.)$ 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($p \to q$) for each edge of the CFG
nodeList = nodes of CFG, in reverse topologically sorted order
for (each $p \in$ nodeList) do
    if ($p$ is a leaf node) then
        valNode($p$) = 1
    else
        valNode($p$) = 0
        for (each edge $p \to q$) do
            valEdge($p \to q$) = valNode($p$)
            valNode($p$) += valNode($q$)
        end
    end
end
```

With edges numbered with $\mathsf{valEdge}(.)$ 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 $\mathsf{valEdge}(.)$ 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 2.3(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 2.3(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 [26] 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 2.3.4, we describe BLAST's compact representation of the whole program path that builds on prior work in this area [66]. 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 2.3(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 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 2.3.6).

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 2.4 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

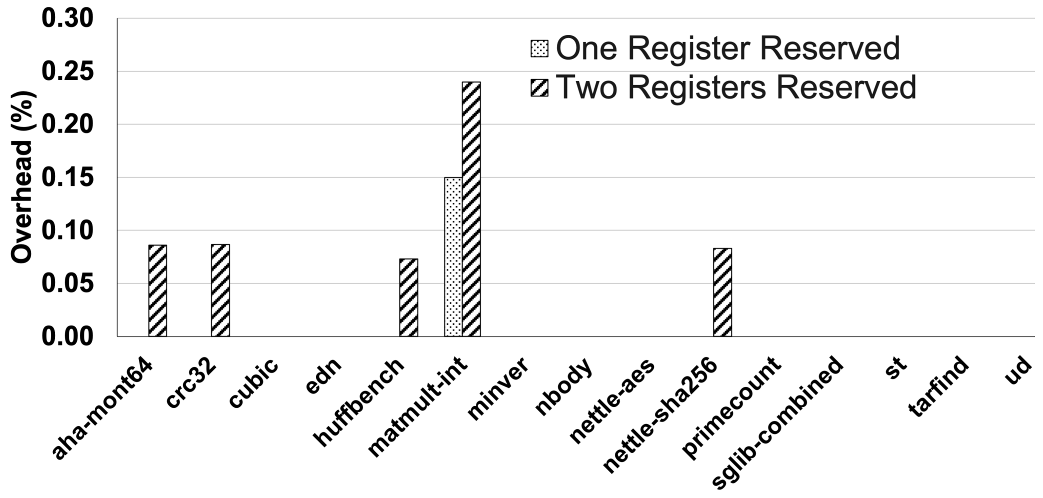


Figure 2.4: Impact of reserving registers for `BLReg` and `LogReg`.

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 2.3(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 [66]), 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). Figure 2.5 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
```

Figure 2.5: Blast's instrumentation inserted before a function call to store the path index and function ID.

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 2.1 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 2.4(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 2.3 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 2.1). The number of log entries reported in Table 2.3 is therefore simply the sum of the corresponding control events reported in Table 2.2.

As Table 2.3 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 2.1) 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,

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 2.3: 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 2.2).

thereby resulting in a total of 348,670,006 log entries, which marginally improves upon OAT (348,840,008 entries, as reported in Table 2.2). Section 2.4 presents detailed results showing the impact of inlining.

2.3.3 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 2.3.1, 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

```
str w8, [x29, #4] // perform the store
add x9, x29, #4 // obtain the store address
and x9, x9, #0x7ffffffffff00000 // mask the store address
ldr x10, log_start_addr // 1MB-aligned log start
subs x9, x9, x10 // if equal, then abort
b.e _abort
```

Figure 2.6: SFI check inserted after a store instruction to ensure that store address does not corrupt the log.

library [76] 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. Figure 2.6 shows the instrumentation inserted after a store instruction assuming a 1MB log.

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 [90], 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 [4] 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.

2.3.4 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 [5]), 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 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* [66]. 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`.

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

WPP grammar:
$$\begin{array}{ccc} \mathsf{S} & \rightarrow & \mathsf{aXdX} \\ \mathsf{X} & \rightarrow & \mathsf{bc} \end{array}$$

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 2.7: 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.

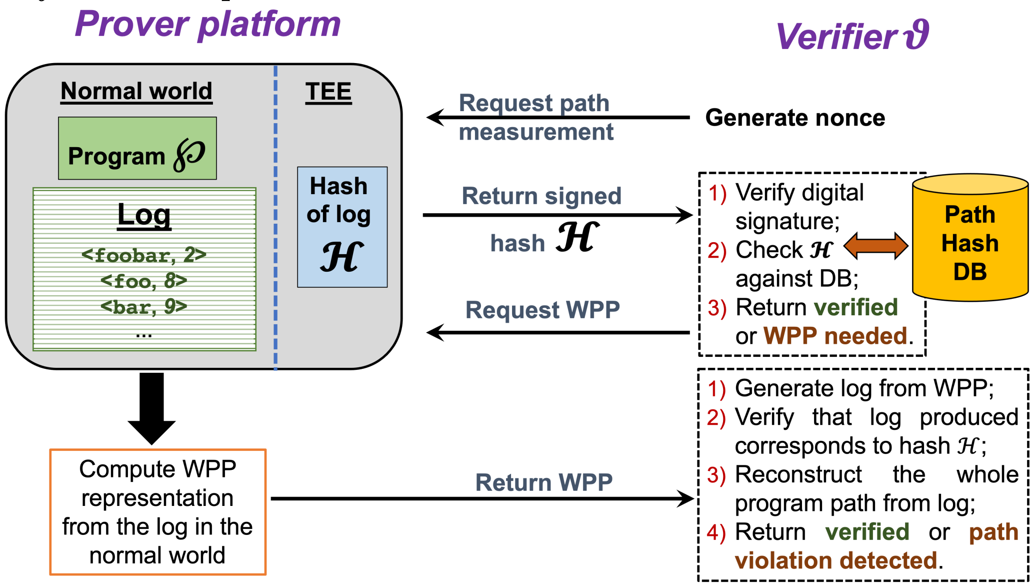

Prover platform
Verifier 𝜗
Normal world
TEE
Program ℘
Log
<foobar, 2>
<foo, 8>
<bar, 9>
...
Hash of log
ℋ
Request path measurement
Generate nonce
Return signed hash ℋ
1) Verify digital signature;
2) Check ℋ against DB;
3) Return verified or WPP needed.
Path Hash DB
Request WPP
Compute WPP representation from the log in the normal world
Return WPP
1) Generate log from WPP;
2) Verify that log produced corresponds to hash ℋ;
3) Reconstruct the whole program path from log;
4) Return verified or path violation detected.

Figure 2.8: Workflow for verification of CFA.

Each such tuple is assigned an identifier that then becomes a terminal symbol in the resulting grammar. The WPP construction algorithm [66] identifies common fragments in the log, and "lifts" them to non-terminal symbols. Consider, for example, a fragment from a log shown in Figure 2.7. This log fragment can be represented symbolically as abcdbc, and its compressed WPP representation would be as shown on the right in Figure 2.7. 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 [66] 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 2.8 summarizes the workflow for path measurement verification.

2.3.5 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 [76] 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.

2.3.6 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 2.1). 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 [30, 40].

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 [69] provide a detailed overview of the security implications of interrupt handling to CFA. Prior work [24, 49] 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 [24, 49]. These methods have been developed in our next work (Chapter 3).

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 [87].

2.4 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 2.4 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 2.9 breaks down the contribution of each of Ball-Larus instrumentation, SFI, and log commit operations to this overhead;

3. Table 2.5 presents the runtime overhead imposed by BLAST for attesting *musl libc* along with $\mathcal{P}$'s execution.

4. Table 2.6 analyzes the energy consumption of BLAST;

5. Table 2.7 quantifies the benefits of WPP for log compression;

6. Section 2.4.5 shows the effectiveness of Blast at detecting anomalous behavior in $\mathcal{P}$ using a case study.

2.4.1 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 2.4(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.

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

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%**

2.4(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.

Embench-IOT Program ↓	**Blast w/o parallelism**	
	Time (s) & Overhead	Waiting Time (s)
aha-mont64	46.10 (296%)	18.5
crc32	35.05 (202%)	17.13
cubic	2.05 (15%)	0.03
edn	80.51 (209%)	35.81
huffbench	48.44 (253%)	23.58
matmult-int	81.51 (143%)	38.01
minver	13.31 (203%)	5.6
nbody	1.26 (142%)	0.45
nettle-aes	26.37 (60%)	5.05
nettle-sha256	16.38 (36%)	3.42
primecount	50.6 (229%)	27.31
sglib-combined	55.96 (232%)	23.61
st	2.32 (190%)	0.9
tarfind	11.51 (206%)	4.74
ud	56.03 (208%)	26.11
Avg. Ovhd.	175%	

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

Embench-IOT Program ↓	**Blast versus CFLAT & OAT**			
	CFLAT time (s)	$\frac{\text{CFLAT}}{\text{Blast}}$	OAT time (s)	$\frac{\text{OAT}}{\text{Blast}}$
aha-mont64	87.12	(3.10×)	40.4	(1.44×)
crc32	105.23	(5.74×)	52.64	(2.87×)
cubic	2.08	(1.03×)	2.03	(1.00×)
edn	111.24	(2.43×)	45.29	(0.99×)
huffbench	99.2	(3.87×)	50.09	(1.96×)
matmult-int	120.68	(2.73×)	43.83	(0.99×)
minver	28.61	(3.59×)	12.34	(1.55×)
nbody	1.90	(2.32×)	0.83	(1.01×)
nettle-aes	23.60	(1.10×)	22.30	(1.04×)
nettle-sha256	22.90	(1.74×)	13.24	(1.01×)
primecount	161.59	(5.68×)	88.63	(3.12×)
sglib-combined	154.74	(4.69×)	83.89	(2.54×)
st	5.30	(3.61×)	2.66	(1.81×)
tarfind	30.68	(4.45×)	15.56	(2.26×)
ud	103.95	(3.36×)	41.79	(1.35×)
Avg.:		3.30×		1.66×

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

Table 2.4: 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.

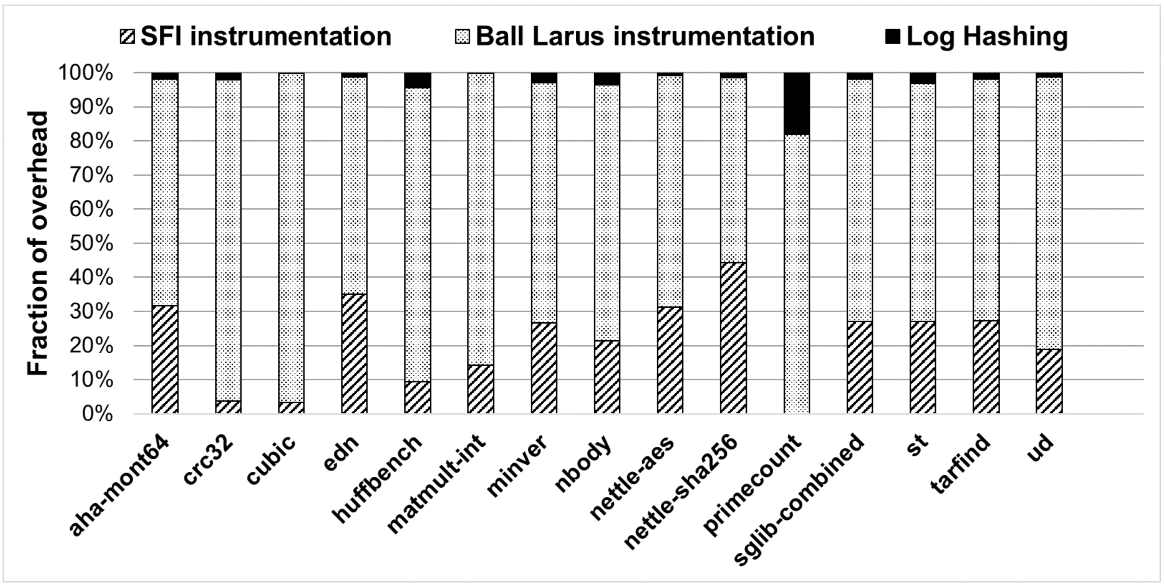


Figure 2.9: Contribution of SFI, Ball-Larus instrumentation, and log commit operations to overhead.

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 CFA 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 2.9 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 of $\mathcal{P}$. Table 2.4(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 2.4(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 as reported in Table 2.4(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 2.3). 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.

2.4.2 Library Attestation with BLAST

We also extend BLAST to attest library code, demonstrating that whole-program attestation, including standard libraries like *musl libc*, is practical with low additional overhead. Libraries are often overlooked during CFA of the program $\mathcal{P}$, even though commonly used libraries such as *libc* have been shown to be attractive targets for control-flow hijacking attacks [83]. This makes it critical to attest the execution of libraries in conjunction with the main program. To

Embench-IOT Program ↓	**Execution Time(s)**				
	#Library Calls	**Baseline**	**Only Program**	**Program + Library**	**(Overhead)**
aha-mont64	1	11.16	24.53	24.73	(0.82%)
crc_32	2	10.7	17.78	17.91	(0.73%)
cubic	20	1.65	1.96	2.32	(18.37%)
edn	5	25.35	44.06	44.57	(1.16%)
huffbench	11	13.38	24.34	25.16	(3.37%)
matmult-int	5	32.64	46.96	47.22	(0.55%)
minver	7	4.3	7.3	7.47	(2.33%)
nbody	6	0.5	0.81	0.83	(2.47%)
nettle-aes	8	16.05	23.3	23.57	(1.16%)
nettle-sha256	11	11.76	17.51	17.96	(2.57%)
primecount	1	15	28.78	28.94	(0.56%)
sglib-combined	4	16.1	30.68	31.06	(1.24%)
st	7	0.76	1.4	1.6	(14.29%)
tarfind	3	3.56	6.53	6.69	(2.45%)
ud	3	17.4	28.62	28.72	(0.35%)
Average overhead:					3.49%

Table 2.5: Performance of library attestation with Blast on Embench-IOT benchmarks. Both the program and the musl libc library are instrumented with Blast instrumentation. Numbers reported are preliminary results.

evaluate this, we apply Blast to the musl libc [46] library. We choose musl libc instead of GNU libc as it is lightweight and allows efficient static linking.

Embench-IOT programs are designed for bare-metal embedded devices and invoke only few library calls as shown in Table 2.5. For these experiments, we use the best-case configuration of Blast for Embench-IOT programs which minimizes runtime overhead by leveraging Ball-Larus instrumentation, local logging with parallel flushing, and selective inlining (Table 2.4 (a)). We statically link musl libc with each main program. Note that the performance numbers for attesting only the main program may differ slightly from previous results due to changes in the evaluation setup, including the underlying library and OP-TEE version.

A key observation is that attesting library code introduces only a modest overhead on top of attesting the main program as shown in Table 2.5. On average, the added overhead is just 3.49%, demonstrating the practicality of extending CFA to libraries. This is encouraging, given that libraries such as musl libc are widely used and often targeted in control-flow hijacking attacks, *e.g.*,, via gadgets in libc.

Most benchmarks show overheads below 3%, when the number of library calls ranges between 3 and 11 (*e.g.*, `matmult-int`, `primecount`, `huffbench`). Only a few programs such as `cubic`

Embench-IOT	**Energy Consumption in Joules**		
Program ↓	**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 2.6: 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.

and `st` exhibit higher overheads (18.37% and 14.29%, respectively), which is attributed to the maths-related library function calls in these workloads. Importantly, the low incremental cost of library attestation reinforces our claim that whole-program attestation, encompassing both application and libraries, is feasible with Blast and it supports the inclusion of library code in the attestation boundary for stronger security guarantees.

2.4.3 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 2.4(a). Table 2.6 reports the results of these experiments, which show an average increase in energy consumption of about 80% across the Embench-IOT benchmarks.

2.4.4 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 2.3 for raw log entry numbers). The size of the resulting log can therefore run into MBs (or even GBs), as shown in Table 2.7. 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 2.3.4, $\mathcal{V}$ 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 2.7 compares the raw size of the log, a version of the log compressed with the `bzip2` utility [82], and the WPP representation of the log. Bzip2 uses 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 2.7 clearly shows the benefits of the WPP representation, which is just a few hundred *bytes* in each case.

2.4.5 Case Study: Open Syringe Pump Benchmark

We evaluated Blast using the Open Syringe pump benchmark [92], 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 2.8, the raw *whole-program* CFA overhead incurred using the Blast-instrumented application is 0.14s

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

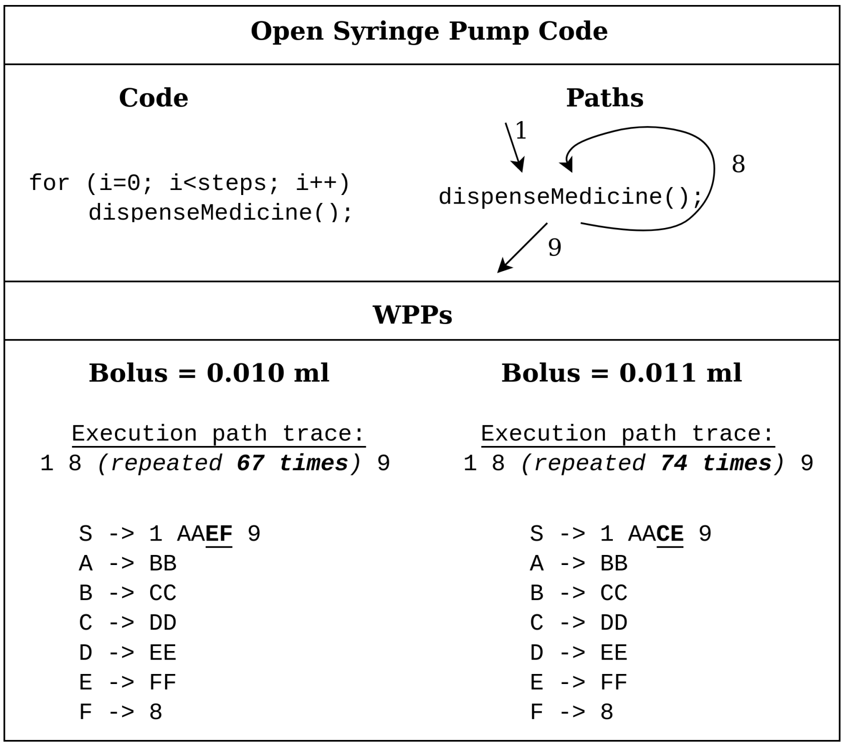
Table 2.7: Comparison of the size of the raw log against a log compressed with two lossless approaches: bzip2 and WPP.

Open Syringe Pump	**Baseline**	**Blast**	
Bolus (ml) ↓	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 2.8: Performance evaluation for different bolus amounts. Bolus (mL) is the dose of drug.

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* [5, 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



```
for (i=0; i<steps; i++)
    dispenseMedicine();
```

Bolus = 0.010 ml

```
Execution path trace:
1 8 (repeated 67 times) 9

S -> 1 AAEF 9
A -> BB
B -> CC
C -> DD
D -> EE
E -> FF
F -> 8
```

Bolus = 0.011 ml

```
Execution path trace:
1 8 (repeated 74 times) 9

S -> 1 AACE 9
A -> BB
B -> CC
C -> DD
D -> EE
E -> FF
F -> 8
```

Figure 2.10: 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 in the ported application. The CFG consists of three acyclic paths: a loop entry path with Ball-Larus value `1`, a loop iteration path with value `8`, and a loop exit path with value `9` as shown in Figure 2.10. 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.

File hashing workloads in Mbed-TLS

Algorithm	File	Baseline	$\mathcal{S}$=1	$\mathcal{S}$=0.1		$\mathcal{S}$=0.01		$\mathcal{S}$=0.001	
	(MBs)	Time (sec)	#WSTs	#WSTs	Time(sec)/Ovhd.	#WSTs	Time(sec)/Ovhd.	#WSTs	Time(sec)/Ovhd.
MD5	5	0.29	435,304	51,212	10.61 (3,558.62%)	5,118	1.47 (406.90%)	528	0.55 (89.66%)
	50	2.95	4,352,104	512,012	105.1 (3,462.71%)	51,204	13.47 (356.61%)	5,132	4.24 (43.73%)
	100	5.9	8,704,104	1,024,012	210.23 (3,463.22%)	102,404	26.82 (354.58%)	10,246	8.34 (41.36%)
SHA256	5	1.76	3,727,552	384,014	77.95 (4,328.98%)	38,404	9.53 (441.48%)	3,842	2.67 (51.70%)
	50	17.59	37,273,792	3,840,014	776.59 (4,314.95%)	384,004	93.94 (434.05%)	38,404	25.52 (45.08%)
	100	35.18	74,547,392	7,680,014	1555.3 (4,320.98%)	768,004	188.09 (434.65%)	76,804	50.88 (44.63%)
SHA512	5	1.2	4,546,913	454,166	91.14 (7,495.00%)	45,362	10.31 (759.17%)	4,548	2.24 (86.67%)
	50	11.98	45,465,953	4,541,462	910.16 (7,497.33%)	454,148	102.03 (751.67%)	45,352	21.2 (76.96%)
	100	23.98	90,931,553	9,082,902	1819.17 (7,486.20%)	908,292	203.89 (750.25%)	90,710	42.27 (76.27%)
RIPEMD160	5	0.74	435,320	51,212	11.11 (1,401.35%)	5,118	1.92 (159.46%)	528	0.99 (33.78%)
	50	7.39	4,352,120	512,012	110.05 (1,389.17%)	51,204	18.01 (143.71%)	5,132	8.73 (18.13%)
	100	14.78	8,704,120	1,024,012	169.45 (1,046.48%)	102,404	35.85 (142.56%)	10,246	17.31 (17.12%)

Encryption/Decryption workloads in Mbed-TLS

Algorithm	Number of	Baseline	$\mathcal{S}$=1	$\mathcal{S}$=0.1		$\mathcal{S}$=0.01		$\mathcal{S}$=0.001	
	operations	Time (sec)	#WSTs	#WSTs	Time (sec)/Ovhd.	#WSTs	Time (sec)/Ovhd.	#WSTs	Time (sec)/Ovhd.
AES	600	0.013	19,679	2,427	0.62 (4,706.20%)	813	0.3 (2,225.58%)	613	0.26 (1,915.50%)
DES	600	0.014	19,935	2,251	0.58 (4,145.97%)	771	0.29 (2,022.99%)	620	0.26 (1,803.37%)
ARIA	600	0.041	161,704	7,509	1.64 (3,921.58%)	1,017	0.35 (758.26%)	375	0.22 (439.48%)

Each of the 600 encryption/decryption operations processes one block of data—AES and ARIA use 16B blocks, while DES uses 8B blocks. Numbers reported are averaged over 10 runs, and the standard deviation was less than 0.2%. The baseline is the time to execute the benchmark without any instrumentation. We report just the #WSTs for the case $\mathcal{S}$=1 (*i.e.*, full tracking/no sampling).

Table 2.9: Performance of the file hashing and encryption/decryption workloads in Mbed-TLS instrumented with sampling, for various values of $\mathcal{S}$.

Embench-IOT	Baseline	$\mathcal{S} = 0.01$			$\mathcal{S} = 0.001$			$\mathcal{S} = 0.0001$			$\mathcal{S} = 0.00001$		
Program ↓	Time (sec)	#WSTs	Time (sec)/Ovhd.		#WSTs	Time (sec)/Ovhd.		#WSTs	Time (sec)/Ovhd.		#WSTs	Time (sec)/Ovhd.	
aha-mont64	11.61	169,203	45.88	(295.18%)	16,923	15.7	(35.23%)	1,695	12.61	(8.61%)	171	12.35	(6.37%)
crc32	11.56	6,970,003	1396.46	(11,980.10%)	697,003	158.78	(1,273.53%)	69,703	35.23	(204.76%)	6,563	22.74	(96.71%)
cubic	1.77	4,003	2.77	(56.50%)	403	2.06	(16.38%)	43	1.95	(10.17%)	7	1.95	(10.17%)
edn	26.02	3,257,283	669.41	(2,472.67%)	592,995	143.44	(451.27%)	58,791	37.79	(45.23%)	5,115	27.18	(4.46%)
huffbench	13.71	4,664,443	934.75	(6,718.02%)	466,447	106.04	(673.45%)	42,430	22.73	(65.79%)	41	13.84	(0.95%)
matmult-int	33.43	2,699	34.18	(2.24%)	1,027	33.78	(1.05%)	861	33.76	(0.99%)	843	33.72	(0.87%)
minver	4.39	682,653	140.02	(3,089.52%)	68,268	18.5	(321.41%)	6,879	6.28	(43.05%)	692	5.03	(14.58%)
nbody	0.52	47,223	9.98	(1,819.23%)	4,725	1.59	(205.77%)	448	0.74	(42.31%)	50	0.662	(27.31%)
nettle-aes	16.48	521,823	120.93	(633.80%)	52,185	27.07	(64.26%)	5,291	17.69	(7.34%)	520	16.76	(1.70%)
nettle-sha256	12.04	23,753	16.75	(39.12%)	2,378	12.85	(6.73%)	240	12.36	(2.66%)	26	12.31	(2.24%)
picojpeg	18.33	4,882,923	995.92	(5,333.28%)	488,295	126.08	(587.83%)	48,856	38.09	(107.80%)	4,863	29.25	(59.57%)
primecount	15.34	2,822,823	572.18	(3,629.99%)	282,285	71.1	(363.49%)	3	15.42	(0.52%)	3	15.45	(0.72%)
qrduino	15.61	1,750,436	363.43	(2,228.19%)	162,917	49.56	(217.49%)	14,688	20.07	(28.57%)	2,470	17.58	(12.62%)
sglib-combined	16.84	2,880,869	591.49	(3,412.41%)	288,089	78.9	(368.53%)	29,142	26.99	(60.27%)	1,983	21.43	(27.26%)
st	0.8	262,083	53.07	(6,533.75%)	15,811	4.52	(465.00%)	1,583	1.69	(111.25%)	161	1.41	(76.25%)
tarfind	3.76	726,623	149.21	(3,868.35%)	72,665	20.41	(442.82%)	7,269	7.38	(96.28%)	729	6.03	(60.37%)
ud	18.14	1,950,963	403.95	(2,126.85%)	195,099	56.9	(213.67%)	19,407	22.15	(22.11%)	1,851	18.61	(2.59%)

We compare the baseline runtime of the benchmark observed on a Raspberry Pi-3 B+ to the runtime of the benchmark with various Bernoulli sampling rates ($\mathcal{S}$). Also shown is the number of TEE world switches observed (#WSTs) for that $\mathcal{S}$. Numbers reported are averages of 10 runs and the standard deviation is <0.2%.

Table 2.10: Performance of Embench-IOT benchmarks when instrumented with sampling.

2.5 Sampling: A Strawman Approach to Reduce Overheads of CFA

Following the approach of prior works [5, 87], which attest only parts of a program's execution, we explored sampling as a means to reduce the overheads of CFA. We show that while reducing the granularity of attestation via sampling does lower performance overheads, it is not sufficient on its own to achieve practical overheads and significantly impacts the security guarantees of the attestation approach.

We adapt the idea of sampling to selectively decide whether to measure the paths in a function. We give the verifier $\mathcal{V}$ control over which parts of program $\mathcal{P}$ are subject to path attestation. While it is possible to implement such control at the granularity of individual domain switch trampolines, we chose to implement it at the granularity of functions. That is, for each function (or at each function call) of $\mathcal{P}$, the verifier can make a choice as to whether to measure the paths of that function (or not). For example, $\mathcal{V}$ can decide not to measure the paths of a function `foo`, but may decide that all the paths of a function `bar` called by `foo` must be measured. The function `bar` may in turn call a third function `zoo` that $\mathcal{V}$ is not interested in measuring. We provide this level of flexibility to control path measurement at function call granularity.

2.5.1 Instrumentation and Verification with Sampling

We integrate sampling into the workflow of CFA. The verifier $\mathcal{V}$ supplies a bit-string $\mathcal{B}$ indicating the function calls in $\mathcal{P}$ that it is interested in attesting. Each slot in $\mathcal{B}$ is reserved for a function in $\mathcal{P}$, and a True (or False) in the corresponding slot denotes that whether the function must be sampled (or not). The prover also stores $\mathcal{B}$ in the TEE as it is included as part of the integrity measurement that it sends in response.

We add instrumentation to $\mathcal{P}$ to copy $\mathcal{B}$ into its address space in the normal world when it starts execution. This step is to avoid performing a world switch to lookup $\mathcal{B}$ from the TEE at each function call. To support sampling, we create two copies of each function $\mathcal{F}$. One copy, $\mathcal{F}_{BL}$, is instrumented in full with Ball-Larus numbering for instrumentation placement, as described in Section 2.3.2. In the second copy, $\mathcal{F}_{None}$, we do not insert instrumentation (either for world switches or operations on `BLReg`), except at any function call instructions in $\mathcal{F}$ (*i.e.,* functions that are called by $\mathcal{F}$). Preceding each function call site in $\mathcal{F}_{None}$, we add instrumentation that consults $\mathcal{B}$ to determine how to handle the called function $\mathcal{G}$. Because this happens just preceding the call site, we can resolve the target of indirect calls as well. If

$\mathcal{B}$ determines that paths in $\mathcal{G}$ must be attested, the instrumentation inserted by us invokes the instrumented copy $\mathcal{G}_{BL}$; else it invokes $\mathcal{G}_{None}$.

Note that there are no world switches during the execution of $\mathcal{F}_{None}$ or $\mathcal{G}_{None}$ because $\mathcal{B}$ is stored in $\mathcal{P}$'s address space. World switches are initiated only if $\mathcal{B}$ determines, say, that paths in $\mathcal{G}$ must be attested. In that case, when $\mathcal{G}$ is invoked in $\mathcal{F}$, the instrumentation in $\mathcal{F}_{None}$ performs a world switch to record the $\mathcal{G}_{BL}$'s address in the TEE, and then calls $\mathcal{G}_{BL}$. When $\mathcal{G}_{BL}$ returns to $\mathcal{F}_{None}$, execution resumes with no further world switches until the next function call for which $\mathcal{B}$ determines that path attestation is required.

At first blush, it may appear that our approach doubles code size, and increases memory pressure on resource-constrained embedded devices. However, note that $\mathcal{B}$ is supplied upfront by $\mathcal{V}$. This allows the prover to selectively load the corresponding function copies into physical memory, while keeping the other copies on disk. This strategy effectively retains the original memory footprint of $\mathcal{P}$ on the prover's platform.

Further, sampling can be configured to either be *targeted* or *probabilitistic*. In targeted sampling, the verifier $\mathcal{V}$ identifies a specific set of functions whose paths must be attested. Our instrumentation ensures that each call to those functions will invoke the fully-instrumented copies of those functions. Our sampling approach discussed thus far assumed targeted sampling. With targeted sampling, the verifier $\mathcal{V}$ only gets the path attestation measurements of functions it specifies in $\mathcal{B}$, and these can be checked with the measurement DB, as before.

We also provide the choice of probabilistic sampling, in which $\mathcal{V}$ does not identify a specific set of functions to attest. Instead, the decision on whether to invoke the fully-instrumented or uninstrumented copy of each function is made at each call site (*i.e.,* an independent Bernoulli trial [75]). Such Bernoulli sampling can be useful if $\mathcal{V}$ wishes to attest paths in some functions of $\mathcal{P}$ (probabilistically) without incurring the high overheads associated with running the fully instrumented version of $\mathcal{P}$. Our approach naturally extends to probabilisitic sampling as well. The verifier $\mathcal{V}$ simply supplies a long bitstring $\mathcal{B}$ each of whose entries is set independently to true or false with a fixed probability $\mathcal{S}$ (as is routine for Bernoulli sampling). $\mathcal{B}$ is parameterized by the Bernoulli sampling probability $\mathcal{S}$, which indicates the probability of each bit being set to True in $\mathcal{B}$. If configured for probabilisitic sampling, our instrumentation at each call site reads the leading bit in $\mathcal{B}$, takes the decision on which copy of the function to call, and left-shifts the $\mathcal{B}$. Thus, the probability of running an instrumented copy of a function is $\mathcal{S}$. The verification algorithm in $\mathcal{V}$ must also be suitably adapted if probabilistic sampling is used. In essence, $\mathcal{V}$ must replay the execution of $\mathcal{P}$ locally using the same bitstring $\mathcal{B}$ supplied in the challenge, and check that the path numbers collected at the prover match those obtained locally. This verifier design is akin to that proposed in OAT, which uses abstract execution of $\mathcal{P}$ to verify the correctness of the collected integrity measurements.

2.5.2 Security Ramifications of Sampling

The main new consideration with sampling is the bitstring $\mathcal{B}$, which we copy into $\mathcal{P}$'s address space, exposing it to the attacker. However, note that if the attacker modifies $\mathcal{B}$, then it directly impacts the integrity measurement. The integrity measurement notes the address of the function invoked, return addresses, as well as path numbers. If the attacker modifies $\mathcal{B}$, paths in a different set of functions will be attested, and the integrity measurements will differ from those expected by $\mathcal{V}$. In targeted sampling, this involves straightforward matching against the measurement DB in targeted sampling. In probabilistic sampling, $\mathcal{V}$ detects attack during replay.

Sampling comes at a cost—it cannot detect attacks that occur in the execution of uninstrumented versions of functions in $\mathcal{P}$. Because these uninstrumented executions do not generate attestation measurements, any control-flow deviations or injected malicious behavior within them remain undetected by $\mathcal{V}$. Sampling weakens the security guarantees of CFA by introducing blind spots in program execution coverage.

2.5.3 Performance with Sampling

We evaluate the performance benefits of sampling by configuring our tool to use probabilistic sampling and present preliminary results. We chose probabilistic sampling for this evaluation because it allows us to evaluate the effect of varying sampling probability $\mathcal{S}$ on runtime performance overhead.

Table 2.9 shows the results of running sampling approach on file hashing and encryption/decryption functionality of Mbed-TLS. The "baseline" column reports the wallclock time to execute the uninstrumented benchmark on our Raspberry Pi 3 Model B+ platform (with an ARM Cortex A53 Quad core 64-bit processor clocked at 1.4GHz and 1GB LPDDR2 SRAM). We report both the number of world-switches encountered, and the corresponding runtime for various sampling probabilities ranging from $\mathcal{S}$=0.1 to $\mathcal{S}$=0.001. Recall that, the sampling rate $\mathcal{S}$ indicates the probability of executing the instrumented copy of a function at each call site. For example, a sampling rate of $\mathcal{S}$=0.1 implies that on average only 10% of function calls are instrumented. We considered various hashing and encryption algorithms, and varied the sizes of the files used as the workload. We report both the baseline execution time of an uninstrumented version, as well as the number of world switches seen with a fully instrumented version (under the $\mathcal{S}$=1 column). As we can see, several of the functionalities we tested in Mbed-TLS result in millions or tens of millions of world switches. With a world switching overhead of 190μsec on our Raspberry Pi 3 B+, running fully instrumented versions of these applications will be impractical. Sampling reduces the number of world switches, thereby reducing the runtime overheads of the workloads to feasible levels (*e.g.,* with Bernoulli sampling rate $\mathcal{S}$=0.001).

Table 2.10 reports the results of using sampling on Embench-IOT benchmarks [1]. We report both the number of world-switches encountered, and the corresponding runtime for various sampling probabilities ranging from $\mathcal{S}$=0.01 to $\mathcal{S}$=0.00001. We observe that even at relatively low sampling rate of $\mathcal{S}$=0.01 (on average only 1% of function calls are instrumented), most benchmarks incur extremely high overheads. `crc32` incurs a runtime overhead of nearly 12,000%, increasing execution time from 11.56s to 1,396.46s. `huffbench` sees an overhead of 6,718%, while `picojpeg` incurs 5,333%. Even benchmarks with shorter baseline runtimes, such as `nbody` and `st`, observe overheads of 1,819% and 6,534%, respectively. As expected, the performance overhead reduces as $\mathcal{S}$ reduces. At $\mathcal{S}$=0.00001, performance overheads become relatively modest for most benchmarks, with many showing overheads under 10%. However, it does so at the cost of significantly weakened security guarantees.

2.5.4 Shortcomings of Sampling

We explored sampling as a technique to reduce the performance overhead of CFA by instrumenting only a subset of the program's execution events. However, our evaluation shows that sampling suffers from fundamental shortcomings that limit its practicality as an attestation strategy.

1. **Fails to provide complete security guarantees.** By design, sampling attests only a fraction of the execution events or function calls in the program. This inherently prevents $\mathcal{V}$ from obtaining a complete attestation of the program's control-flow path. For applications requiring assurance over the entire execution, such partial attestation is inadequate, as it fails to provide strong guarantees about uninstrumented regions of the program.

2. **Does not sufficiently reduce performance overheads.** While sampling aims to limit instrumentation overhead by reducing the number of monitored events, our results demonstrate that overheads remain impractically high for many programs even at extremely low sampling rates. For instance, with $\mathcal{S}$=0.01, benchmarks such as `crc32` and `huffbench` incurred overheads exceeding 6,000–12,000%, and even at $\mathcal{S}$=0.00001, `crc32` still showed nearly 100% overhead. Thus, sampling alone is insufficient to achieve practical performance for CFA.

In summary, sampling reduces the security guarantees by failing to cover the whole-program execution and does not sufficiently reduce performance costs, making it unsuitable as a standalone approach for scalable CFA.

2.6 Summary of BLAST

Whole program CFA has proven to be an elusive goal. This paper showed that prior methods for CFA 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 CFA with runtime performance overheads of 67% on a set of embedded benchmarks.

Chapter 3

Non-Bare-Metal User-Space Control-Flow Attestation

3.1 Motivation

CFA assumes that the path measurements of $\mathcal{P}$ are integrity-protected as they are recorded and communicated to the TEE. Instrumentation in $\mathcal{P}$ invokes the TEE to either store path measurements as soon as an event of interest is recorded [5, 87] (*e.g.,* when a branch is taken) or caches them in a CFA log and periodically flushes them to the TEE as was done in BLAST (Chapter 2). The latter case vastly improves the performance of CFA, but requires protecting the state of the log from unauthorized modification. This is enabled in $\mathcal{P}$ via either software-fault isolation (SFI) [90] or hardware support (*e.g.,* Intel MPK [57, 101]).

Section 3.2 shows that in non-bare-metal settings, the security assumptions made by prior CFA methods do not hold. Attacks on the REE OS may compromise $\mathcal{P}$'s TEE invocations and/or the corrupt the measurements cached in the CFA log. These attacks completely subvert the guarantees offered by CFA, thereby greatly limiting the scenarios where it can be securely used. While the assumption of bare-metal execution may hold for specialized embedded devices, it does not hold for higher-end devices. For example, many embedded devices, robots and even satellites [45] contain powerful processors and allow concurrent execution of multiple applications. The presence of an REE OS greatly eases application development and isolation by enabling abstraction and virtualization of the hardware. Blindly trusting this REE OS, as prior non-bare-metal CFA approaches [7, 89, 101] have done, risks bringing a large, potentially vulnerable OS code base into the TCB [24, 62]. System-level adversaries with control over this REE OS can completely subvert the guarantees required for CFA to operate securely.

The main contribution described in this chapter is **Sulfur**, a system architecture that enables robust CFA of user-space applications in non-bare-metal settings. We use the term *robust* to mean that the program trace recorded by CFA is an accurate record of the user-space program $\mathcal{P}$'s execution—see Section 3.5 for a formal treatment of robustness. CFA methods are able to record correct information regarding program execution paths only when certain assumptions regarding the execution environment hold. SULFUR achieves this goal via a co-developed REE OS, SULFUROS, atop which the CFA measurements collected during the execution of a user-space program $\mathcal{P}$ remain secure despite the presence of a system-level adversary (that may have hijacked SULFUROS at runtime). The co-developed REE OS incorporates defenses to protect critical components in the REE OS and user space from adversaries. Having a secure foundation in the REE allows SULFUR to protect the assumptions required for CFA and robustly attest the execution of user-space programs in non-bare-metal settings. Our SULFUROS prototype, which targets the AArch64 architecture and is based on Linux, accomplishes this goal using AArch64's privileged-access never (PAN) [13] and privileged-execute never (PXN) [14] to ensure that SULFUROS cannot access data or execute instructions in $\mathcal{P}$'s address space. Similar features exist on x86 CPUs as well—supervisor-mode access prevention (SMAP) and supervisor-mode execution prevention (SMEP) [57].

SULFUROS uses an approach akin to paravirtualization [24, 27] to ensure that any security-critical changes that can potentially violate the guarantees required for CFA, *e.g.*, modifications to system-control registers, are shepherded via the TEE where a trusted SULFUR MONITOR performs these modifications after suitable security verification. SULFUROS is compiled with control-flow integrity (CFI) protection [3] enabled using AArch64's branch-target instructions (BTI) and pointer-authentication codes (PAC) [77]. Once SULFUROS' image is attested and verified at boot-time by TEE, CFI restricts the power of any run-time attacks that can be launched by an adversary against SULFUROS. Section 3.3 and Section 3.4 describe the design and implementation of SULFUR and SULFUROS.

Our evaluation shows that SULFUR imposes only minimal additional runtime performance overhead on existing CFA systems. For example, using the Embench-IoT embedded benchmark suite [1], we found that SULFUR imposes an average additional overhead of just 1.95% (on average) for CFA-based attestation of a program, compared to CFA-based attestation atop a commodity Linux OS. This overhead primarily stems from the additional security checks performed within SULFUROS and the TEE to protect the security state of a program $\mathcal{P}$ as it is traced using CFA's mechanisms. As SULFUR enables non-bare-metal execution of $\mathcal{P}$, *i.e.*, atop an OS, other programs that are not CFA-instrumented can also execute concurrently with $\mathcal{P}$—a feature not easily possible in bare-metal settings. Such programs suffer a runtime overhead of just 1.73% on average when running atop SULFUROS, again using the Embench-IoT benchmark suite.

To summarize the main contributions described in this chapter:

- We show that prior methods for CFA, which have focused on bare-metal settings, are insecure when applied to programs that run in non-bare-metal settings (Section 3.2).

- We present the design (Section 3.3) and implementation (Section 3.4) of SULFUR, a system that securely enables user-space CFA in non-bare-metal settings. SULFUR allows robust CFA measurement of a user-space program $\mathcal{P}$ that runs on top of a customized OS—SULFUROS—that leverages hardware support (PAN and PXN) to protect CFA state before it is committed to the TEE.

- Our evaluation of SULFUR (Section 3.6) with various benchmarks shows that it accomplishes its goals without excessive runtime performance overheads.

3.2 System-Level Adversaries and CFA

Most prior CFA methods operate under the assumption of a bare metal setting, where the program $\mathcal{P}$ being attested runs directly on the hardware in privileged mode. $\mathcal{P}$ interfaces with hardware via interrupt service routines (ISRs) and CFA approaches attest the execution of the main program $\mathcal{P}$. ISRs execute non-deterministically because they are invoked whenever interrupts arrive. To ensure the integrity of CFA during ISR execution, $\mathcal{P}$'s context must be saved and restored from the TEE [69]. Alternatively, the execution of the ISR itself can be traced with CFA [87]. However, because ISRs execute non-deterministically, CFA measurements of ISRs must be stored separately from $\mathcal{P}$'s CFA measurements as non-determinism complicates the task of the verifier $\mathcal{V}$. CFA techniques primarily focus exclusively on control-flow behavior, but can be supplemented to record sensitive data values as well [87, 93]. Because only one application runs in this setting, attesting $\mathcal{P}$ suffices to ensure execution integrity, and the TCB is typically the TEE, consisting of a secure-boot mechanism and a security monitor.

In the few instances when CFA has been applied in non-bare-metal settings [7, 89, 101], the focus has been on scale and performance. These methods have extended CFA to efficiently attest complex user-space applications. However, these methods fundamentally trust the underlying OS, which is therefore part of the TCB. Including the large, vulnerable code-base of the OS in the TCB is fraught with risks. A vulnerable OS can be compromised (*e.g.,* with kernel-level rootkits) and as we describe below, can subvert many assumptions that CFA requires for security.

System-Level Adversaries in Non-Bare-Metal Settings. In non-bare-metal settings, CFA relies critically on the underlying REE OS to be trustworthy, failing which CFA cannot be trusted to robustly record the execution of a user-space program $\mathcal{P}$. Traditional CFA methods are robust against application-level adversaries that tamper with $\mathcal{P}$'s control-flow, *e.g.,* by modifying condition variables, code pointers and return addresses. However, they do not work if the REE OS is under the control of a *system-level adversary,* **SysAdv**. Such an adversary in the OS can compromise the foundational requirements for CFA of $\mathcal{P}$ *viz.,* integrity of $\mathcal{P}$'s code, execution state, and the CFA measurement generated by the CFA mechanism, as described below. Moreover, unlike application-level adversaries, whose actions will leave a detectable trace in the CFA measurement, SysAdv can subvert CFA stealthily by corrupting the execution state of $\mathcal{P}$. The measurement report in the presence of SysAdv can therefore no longer serve as a robust record of $\mathcal{P}$'s execution. The presence of SysAdv compromises the following key foundational requirements required for CFA:

- **[R1] Write eXecute Never (WXN):** Disabling WXN enables code injection or modification in $\mathcal{P}$. Without exception, all CFA approaches assume that WXN is enabled (also colloquially called data-execution prevention), and all of them are subverted by violating this invariant.

- **[R2] Secure communication with the TEE:** All interactions between $\mathcal{P}$ and TEE pass through the REE OS, which the SysAdv can intercept or manipulate, thereby corrupting the CFA measurement.

- **[R3] Integrity of execution state during interrupts:** SysAdv can alter return addresses saved during interrupts and context switches, causing the application to resume execution from incorrect locations. Some CFA approaches use dedicated registers to store CFA state (Chapter 2, [7]), and SysAdv can corrupt those registers as well. SysAdv, with full control over the REE OS, can control when interrupts occur and use them to stitch together gadgets that can bypass CFA instrumentation in $\mathcal{P}$ entirely. Such attacks have even been demonstrated in bare-metal environments through ISR compromise [69], showing that even minimal underlying software layers can be exploited. A full-fledged REE OS presents a much larger attack surface with multiple threat vectors available to SysAdv.

- **[R4] Integrity of CFA$_{Log}$:** Prior methods have suggested that CFA measurements can be cached in $\mathcal{P}$'s address space (CFA$_{Log}$) using suitable methods to isolate them from $\mathcal{P}$ (*e.g.,* Chapter 2, [74, 93, 101]) or in a kernel-protected region [89]. This approach has enabled massive efficiency gains, but the integrity of CFA_{Log} can be compromised by SysAdv.

Consequently, CFA mechanisms in $\mathcal{P}$ cannot be considered robust in the presence of a system-level adversary that compromises the REE OS, thereby violating the above assumptions. Additional protections are required to ensure that the CFA measurements are recorded securely and accurately reflects $\mathcal{P}$'s true control-flow execution.

Why Not Apply CFA to the REE OS? Given these threats, it is natural to ask why CFA methods cannot be extended to the REE OS code also, thereby providing comprehensive system-level measurement. There are two reasons why this approach is challenging:

1. **Performance overheads.** By design, CFA records the proof of execution of a program $\mathcal{P}$ on input $\mathcal{I}$. This requires CFA to trace every aspects of a program $\mathcal{P}$'s control-flow, thereby making it a performance-intensive technology. Early work [5, 6, 87] only applied it to fragments of $\mathcal{P}$, but subsequent optimizations have reduced the overhead of running CFA on whole programs to between 42%-67% across various benchmarks (Chapter 2,[74, 101]). That said, extending CFA to the REE OS code base would impose significant performance overheads and result in vast measurement logs. For example, we estimated that the execution of a *single* `write` system call to commit a 1MB file to disk would result in about 415,000 control-flow events being added to the measurement log. Executing the attendant instrumentation to write these measurements, and protect CFA_{Log} if measurements are cached in the REE, results in large performance penalties.

2. **Non-determinism in the OS.** Support for concurrency, external I/O and interrupts are sources of non-determinism in an OS. CFA tracing records control-flow events of interest in a measurement log that a $\mathcal{V}$ must then be able to interpret and verify. It is challenging to interpret entries in the measurement log with multiple sources of non-determinism. Even in bare-metal settings with a single program $\mathcal{P}$ executing, the CFA traces of ISRs must be recorded separately [87] to allow easy interpretation of the log to trace $\mathcal{P}$'s execution. These problems will exacerbate in a full-fledged REE OS.

Sulfur addresses these challenges by exploring a novel point in the design space. Our co-developed REE OS, SulfurOS incorporates CFA-centric defenses using PAN/PXN in AArch64 to ensure that user-space CFA state cannot be accessed by SysAdv. SulfurOS incorporates safeguards to protect its own runtime integrity from SysAdv. Its execution is constrained by CFI protection using AArch64's PACBTI [20] to restrict the power of control-hijacking attacks, and by TEE-mediated security checks [24, 49] that restrict its ability to modify its own state.

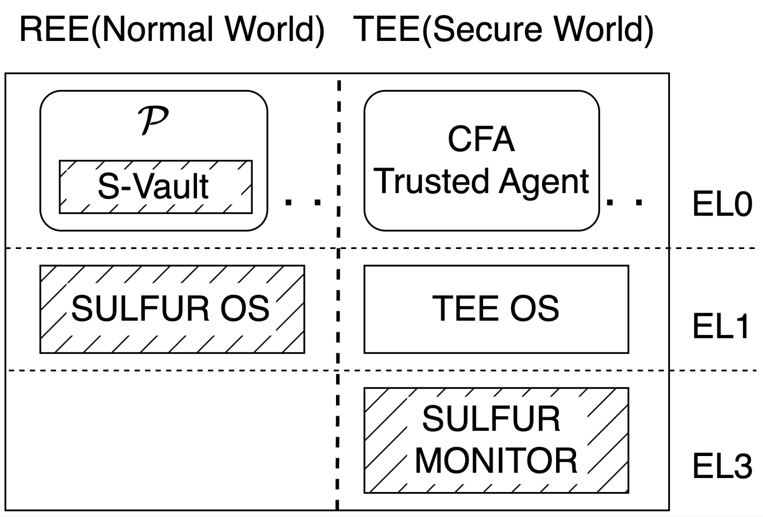
3.3 SULFUR: Assumptions and Design

Before presenting the threat model and assumptions of SULFUR, we first outline the traditional CFA threat model and assumptions. This separation clarifies how SULFUR extends the standard CFA setting and helps situate SULFUR within the prior literature.

Traditional CFA Threat Model and Assumptions. The prover platform is assumed to be equipped with a hardware-based TEE, such as the ARM TrustZone. The adversary typically includes a software attacker who can manipulate user-space code or data but cannot compromise the trusted component (*e.g.,* the TEE or TPM). Prior works generally assumes either a bare-metal environment, where the program runs directly on hardware, or a trusted OS in the REE, which isolates the attested program from untrusted entities. The platform performs a secure boot process rooted in the hardware, ensuring that only verified code executes in both the REE and the TEE.

Goal and Scope of Sulfur. SULFUR aims to ensure that the CFA measurement collected when $\mathcal{P}$ executes $\mathcal{I}$ is an accurate evidence of $\mathcal{P}$'s execution. With SULFUR, this evidence is robust even in the presence of SysAdv in the REE. See Section 3.5 for a formal definition of *robustness* of CFA. Our focus is on control-flow attacks on $\mathcal{P}$, though it can also be extended to record data values of interest in a manner similar to prior work [87, 93]. Unlike prior works that either assume a bare-metal setting or a fully-trusted REE OS, SULFUR targets a practical middle ground—it operates in non-bare-metal settings with SULFUROS—an REE OS that is specifically co-developed to ensure that CFA artifacts cannot be compromised by SysAdv. In SULFUR, CFA measurements only record the user-space operations of a program $\mathcal{P}$. $\mathcal{P}$ may invoke system calls to SULFUROS, but in-kernel program paths followed by system calls are not recorded in CFA measurements. We rely on CFI and other system integrity guardrails to ensure that system calls proceed to completion. Successful completion of I/O by peripherals is beyond the scope of CFA and SULFUR.

Assumptions. Similar to the traditional CFA settings, SULFUR requires the prover platform to be equipped with a TEE, *viz.,* the ARM TrustZone in our prototype. $\mathcal{P}$ runs as a user-mode process in the REE and is managed by the REE OS. We assume that the TEE, *i.e.,* the secure world in ARM Trustzone, cannot be tampered with by any software running in the REE, including the REE OS. The prover platform is booted securely, with a hardware root of trust storing boot-time measurements, and the memory is assumed to have been configured properly



SULFUR's components (shown with hatching), include S-Vault and SULFUROS in the normal world (*i.e.,* the REE) and SULFUR MONITOR in the secure world (the TEE). Exception level (abbreviated EL) is the privilege level in ARM, higher values represent higher privilege. The TEE contains a trusted user-space application (abbreviated TA) that interfaces with $\mathcal{P}$ in the REE to collect and store path measurements. SULFUR's TCB includes SULFUR MONITOR and the components executing in the TEE.

Figure 3.1: Overview of Sulfur's architecture.

at boot time by the TEE. This boot process ensures that the correct OS image is loaded for execution within the REE, *i.e.,* a boot-time static measurement of this OS is recorded in the TEE. We additionally also assume that the AArch64-based prover platform supports PAN [13], PXN [14], and PACBTI [20], as introduced in ARMv8.3-A and above.

Threat Model. SULFUR operates under the assumption of a system-level adversary SysAdv in the REE. SULFUR's TCB consists of SULFUR MONITOR and the components running in the ARM secure world. The TEE measures and boots SULFUROS in the REE, whose code incorporates system-integrity guardrails and CFA-specific mechanisms, to ensure that CFA works as intended on user-space programs in the REE. We disallow dynamic loading of kernel modules after SULFUROS has booted, which prevents introduction of any new kernel-mode code after boot time. SysAdv may attempt to exploit vulnerabilities in SULFUROS or user-space to violate CFA requirements (*i.e.,* R1-R4 in Section 3.2), but user-space CFA must still be robust. SULFUR inherits all the standard assumptions made about a user-space program $\mathcal{P}$ in bare-metal CFA, *e.g.,* that $\mathcal{P}$ is instrumented suitably for CFA and its image is attested before it executes. In non-bare-metal settings, CFA relies on the REE OS to provide features such as WXN, facilitate interaction with the TEE and to set up a secure region for CFA$_{Log}$. SULFUR's CFA-specific mechanisms securely enable these features in the presence of SysAdv.

3.3.1 Overview of SULFUR

Figure 3.1 illustrates the key components of SULFUR on the prover platform. SULFUR introduces a set of carefully engineered defenses that protect CFA from SysAdv in the REE. These defenses (Table 3.1) are broadly classified as:

- **System integrity guardrails:** These guardrails are incorporated into SULFUROS to ensure that security-sensitive operations, *e.g.,* changes to system configuration or SULFUROS's page tables, are redirected to and shepherded by SULFUR MONITOR. SULFUROS is compiled with CFI, limiting the power of kernel code-reuse attacks by SysAdv.
- **CFA-centric protections:** These mechanisms safeguard the execution state of $\mathcal{P}$, CFA-related artifacts, *e.g.,* CFA_{Log}, and REE/TEE communication from SysAdv.

System integrity guardrails aim to protect both the integrity of the OS and the critical state of the system from SysAdv. Guardrails inserted at security-sensitive locations in the SULFUROS kernel invoke the TEE. These invocations are akin to paravirtualization [27]—they invoke SULFUR MONITOR via an ARM *secure monitor call* (`SMC`), which in turn invokes a trusted agent in the secure world. This trusted agent ensures that key system configurations are not tampered (*e.g.,* that WXN remains enabled), prevents any compromise of SULFUROS and application code integrity, and mediates all changes to SULFUROS page tables. This approach is inspired by Knox [24, 49], and ensures the integrity of SULFUROS system state and code throughout the lifetime of the system, thus enabling other defenses to operate reliably. SULFUROS is hardened against control hijacking by enabling PACBTI-based CFI [20] within the kernel. PACBTI-based CFI is now deployed in most commodity compilers. Section 3.4.2 describes the implementation of these guardrails.

SULFUR's CFA-centric protections are a novel set of mechanisms that focus on securing user-space CFA artifacts using hardware support. The centerpiece of these protections is a dedicated memory region called S-Vault (Section 3.3.2) in the address space of the program $\mathcal{P}$ being traced by CFA. S-Vault serves as a storage space for three CFA artifacts: ① as a cache of CFA measurements, *i.e.,* CFA_{Log}, ② to store $\mathcal{P}$'s context during entry/exit to the SULFUROS kernel, and ③ to hold arguments during TEE invocations to store CFA_{Log}. Although S-Vault resides in user-level, SULFUR protects it from SysAdv using PAN. Specifically, PAN disallows privileged code (thus SysAdv) from accessing user-space addresses, and memory load/store operations in SULFUROS's code are restricted by PAN (with a few exceptions). Section 3.4.1 details the implementation of these protections.

Threats to —	Protected by —	Refer —
System Integrity Guardrails		
REE system configuration	SULFUR MONITOR	§3.4.2.1
SULFUROS code integrity	SULFUR MONITOR	§3.4.2.2
SULFUROS control-flow	PACBTI-based CFI	§3.4.2.2
CFA-Centric Protections		
$\mathcal{P}$'s code integrity	SULFUR MONITOR	§3.4.1.1
$\mathcal{P}$'s execution context	SULFUROS gates & S-Vault	§3.4.1.2
CFA$_{Log}$	S-Vault	§3.4.1.3
Attestation call	State bit & S-Vault	§3.4.1.4

Table 3.1: Summary of Sulfur's protections.

Together, these two sets of defenses enable SULFUR to offer robust user-space CFA measurements. They isolate and protect $\mathcal{P}$' execution state and its CFA measurements, ensuring that any deviation in $\mathcal{P}$'s control flow, whether due to attacks in $\mathcal{P}$ or SULFUROS, either cannot modify or leave a detectable attack trail in CFA measurements. SULFUR's use of hardware features (PAN/PXN/PACBTI) allows the defenses to be lightweight.

3.3.2 S-Vault: A Protected, User-Space Region for CFA

At the core of SULFUR's design is S-Vault, a protected user-space memory region for CFA. For a user-space program $\mathcal{P}$ that is to be attested via CFA, SULFUR MONITOR allocates space for S-Vault in $\mathcal{P}$'s address space. We choose to create S-Vault in $\mathcal{P}$'s address space instead of the TEE because that approach would then require an expensive REE/TEE transition for each access to S-Vault. Prior work ([89, 93], Chapter 2) has shown that that minimizing REE/TEE transitions is key to reducing performance overheads of CFA. SULFUR uses AArch64's PAN and SFI to protect S-Vault from unauthorized accesses in the REE.

PAN [13] is a security feature introduced in ARMv8.1-A that prevents privileged OS code (executing at EL1) from accessing user-space memory (EL0) using standard load/store instructions (*i.e.,* `LDR`/`STR`). With PAN enabled, user-space memory access from the OS results in a fault. To access user memory intentionally, the OS must use PAN-aware instructions, called unprivileged load (`LDTR`) and store (`STTR`) instructions, which are checked against EL0 page permissions, even when executed in privileged mode. PAN is enabled by setting the PAN bit in the `SCTLR_EL1` system control register. SULFUROS is designed to only use `LDR`/`STR` instructions for all kernel memory accesses. This ensures that user-space memory, including S-Vault, remains inaccessible to SULFUROS.

However, there are situations in which SULFUROS does legitimately need to access user-

```
and x3, addr, #0x7ffffffffffff000 // mask store addr
cmp x3, =S-Vault_addr // compare masked store address
b.e abort // Branch to abort if equal
sttr reg, [addr] // store instruction
```

Figure 3.2: SFI check inserted before a STTR instruction in a user-access function.

space memory regions, for example, to read/write data from user-space for I/O. In fact, OS kernels have dedicated user-access functions, *e.g.,* `get_user`, `put_user`, `copy_from_user`, `copy_to_user`, `strncpy_from_user`, that are used for this purpose. Such functions in SulfurOS use `LDTR`/`STTR` instructions, suitably protected using SFI as described below, to access user-space application buffers.

SFI-protection in SulfurOS. User-access functions in SulfurOS that have the `LDTR`/`STTR` instructions must not be allowed to modify S-Vault. To ensure this, SulfurOS incorporates SFI checks on `STTR` instructions within these functions. Figure 3.2 shows SFI checks inserted before a `STTR` instruction in `copy_to_user`, with the mask-based checks ensuring that the store does not target the memory region allocated to S-Vault.

The `STTR` instruction writes the value of the register `reg` to the address `addr`. The size of S-Vault is configurable for a given platform. In the example above, we illustrate the SFI instrumentation in SulfurOS assuming that the size of S-Vault is 4KB (*i.e.,* one page). It is allocated at a fixed virtual address within each CFA-traced user process $\mathcal{P}$'s virtual address space by Sulfur monitor before execution. The instrumentation above applies an address mask to the store address to obtain a page-aligned memory address. Address masking based SFI is possible when the size of the S-Vault is a multiple of the page size, and is generally more efficient than performing address range checks [88, 90]. The masked address is then compared against the fixed start address of the S-Vault, provided as an immediate operand value in the `cmp` instruction. If the two values are equal, it indicates that `addr` is within S-Vault. The abort code terminates $\mathcal{P}$ and in turn the collection of CFA measurement; the attack attempt will thus be visible to $\mathcal{V}$.

The above SFI instrumentation is secure only if arbitrary control transfers are restricted in SulfurOS. In particular, control-hijacks in SulfurOS must not be able to bypass the SFI checks before `STTR`. We ensure this by enforcing PACBTI-based CFI in SulfurOS. PACBTI is a security extension introduced in the Armv8.6-A architecture. It combines two complementary mechanisms, BTI and PAC, to defend against control-flow hijacking attacks. BTI inserts special instructions to identify valid landing pads and provides forward-edge CFI. BTI instructions (`bti` c/bti j) serve as designated control-flow landing pads. When BTI is enabled, the processor ensures that indirect branches can only target these landing pads, failing which the hardware raises an exception. PAC provides pointer authentication using reserved upper-order bits in a 64-bit pointer to store a signature (using new AArch64 instructions). This signature, called a pointer authentication code, is formed from the pointer address itself, a cryptographic key (stored in dedicated 64-bit system registers), and a modifier (*e.g.,* stack pointer). PAC is used to sign and verify pointers, *e.g.,* return addresses, function pointers. For example, at the start of a function, the return address in the `LR` register is signed, and is authenticated using the signature before returning. If the check fails, the hardware throws an exception, thereby ensuring that PAC provides backward-edge CFI.

By enabling PACBTI in SulfurOS and protecting the PAC keys stored in system registers (Section 3.4.2.1), we ensure that SysAdv cannot arbitrarily redirect control flow to instructions in the middle of a basic block. In particular, by ensure that no BTI landing pad exists between the SFI check and the corresponding `STTR` instruction, any potential control-flow hijacks within SulfurOS cannot bypass the SFI check and thus protecting S-Vault from SysAdv.

SFI-protection in $\mathcal{P}$. Protecting S-Vault from unauthorized access within the CFA-traced program $\mathcal{P}$ is also crucial. We add SFI checks in $\mathcal{P}$ to prevent unauthorized accesses to S-Vault, as described in prior work (Chapter 2, [74]).

These methods, which cache CFA measurements in the process address space (CFA$_{Log}$), create an isolated memory region for storing measurements using SFI. In Sulfur, we place CFA$_{Log}$ in S-Vault and use SFI address range checks with store instructions in $\mathcal{P}$ to safeguard S-Vault. $\mathcal{P}$ can only access S-Vault through CFA instrumentation explicitly added to store path measurements. A key difference between the SFI instrumentation in $\mathcal{P}$ and the instrumentation in SulfurOS shown earlier is the placement of the SFI checks. Using the same trick as in Chapter 2, the SFI check is placed *after* the store instruction, unlike conventional SFI. This is necessary because we do not require $\mathcal{P}$ to be CFI-protected, as is standard in most prior CFA work (except CFA+ [7] which was the first to combine CFI and CFA). In the absence of CFI, control can be hijacked to execute the store instruction. Placing the SFI check after the store ensures that it will necessarily execute afterwards and detect attempts to target S-Vault. S-Vault also doubles as the storage space for arguments to TEE calls from $\mathcal{P}$ (to flush CFA$_{Log}$) and to store $\mathcal{P}$'s execution context during system calls, context switches and other interrupts.

3.4 Implementation of SULFUR

Our prototype targets an AArch64 TrustZone-based prover platform equipped with PAN/PXN/-PACBTI. The REE (normal world) of this platform runs SULFUROS, which is built as a general-purpose OS by modifying Linux-6.7.0. It can execute arbitrary user-space processes, only some of which may be instrumented for CFA. For the TEE, our prototype uses OP-TEE [76], which runs OP-TEE OS as the TEE OS, and ARM Trusted Firmware [17] as the secure monitor running at EL3. We implement SULFUR MONITOR as a security service in the secure monitor, running at EL3. SULFUR MONITOR initializes platform-specific configurations–creating its page tables, enabling the MMU, and setting up runtime services for handling `SMC`s. It verifies `SMC` validity and directs control to the specified `SMC` handler routine [21]. After booting the TEE OS (in the secure world), SULFUR MONITOR initializes the processor context for REE (normal world) boot and transitions control to the SULFUROS at the highest available exception level (EL2/EL1). SULFUROS incorporates `SMC` hooks that facilitate seamless control transfer to SULFUR MONITOR at strategic points to protect system state and OS integrity as discussed in Section 3.4.2. We first discuss the main contributions of SULFUR, which focus on ensuring robust CFA of $\mathcal{P}$.

3.4.1 CFA-Centric Protections

Software-based CFA approaches require $\mathcal{P}$'s code integrity to be protected. SULFUR ensures the integrity of $\mathcal{P}$'s code and thus the associated CFA instrumentation. In addition, SULFUR protects the integrity of $\mathcal{P}$'s execution context, CFA measurements stored in the REE (*i.e.,* CFA_{Log}, dedicated registers) and attestation calls made to the TEE via SULFUROS. Recall that SULFUROS is a full-service OS on which a program $\mathcal{P}$ being CFA-attested can co-exist with other programs that are not being attested. SULFUR distinguishes a program $\mathcal{P}$ to be CFA attested from other programs using a unique identifier that we add to the binary header of $\mathcal{P}$ (*e.g.,* when $\mathcal{P}$ is being instrumented for CFA).

3.4.1.1 Protecting Code Integrity of $\mathcal{P}$

To ensure the integrity of $\mathcal{P}$'s code, SULFUR: ① verifies the hash of $\mathcal{P}$'s binary at startup, and ② protects the integrity of $\mathcal{P}$'s code at runtime.

1. **Binary verification and initialization.** SULFUROS' process creation code (`exec()`) invokes the SULFUR MONITOR to determine whether the binary is to be attested with CFA. If so, the SULFUR MONITOR records the binary's hash in the TEE. Since SULFUROS

is restricted from directly modifying security-critical system state such as the `TTBR0_EL1` register (which is the process page table base register in AArch64), it delegates the update to the Sulfur monitor. Before setting this register, the Sulfur monitor performs a page table walk to enforce memory safety guarantees: ⓐ code pages must be mapped as read-execute (RX) only; ⓑ data pages must be read-write (RW) only; and ⓒ no physical page may already be used elsewhere with conflicting permissions. Any violation results in immediate termination of the process.

Once the verification succeeds, the Sulfur monitor updates its internal list of physical frames and permissions, and safely sets the `TTBR0_EL1` register. It also sets a reserved bit in `TTBR0_EL1` to denote that the process is being traced via CFA. This flag allows the Sulfur monitor and SulfurOS to distinguish CFA-traced processes from others, supporting concurrent execution of both types of processes. These steps are performed for every new process. All future writes to `TTBR0_EL1` by SulfurOS (*e.g.,* during context switches) results in a call to Sulfur monitor, ensuring only verified page tables are activated. This prevents attacks that attempt to use duplicate page tables with altered access permissions.

2. **Protecting $\mathcal{P}$'s code integrity.** To preserve the integrity of $\mathcal{P}$'s code throughout execution, Sulfur monitor enforces strict control over the modifications of the page table. It write-protects the page tables of $\mathcal{P}$. Any attempt to allocate memory or change permissions, which causes page faults triggers a request to Sulfur monitor, which checks that new mappings do not violate the original protection rules or reuse physical pages in unsafe ways. By delegating memory management to a trusted agent outside SulfurOS, Sulfur prevents runtime code injection or permission tampering.

3.4.1.2 Protecting Integrity of $\mathcal{P}$'s Execution Context

Ensuring integrity of $\mathcal{P}$'s code alone is insufficient to guarantee the integrity of its execution. In non-bare-metal settings, context switches between processes and interrupts that transfer control to the OS are routine. The kernel saves $\mathcal{P}$'s context on the kernel stack and continues servicing the interrupt/system call request. The guarantees of CFA can be completely subverted if $\mathcal{P}$'s context is modified by SysAdv, *i.e.,* without leaving a trace in $\mathcal{P}$'s attestation log. Sulfur's design protects the integrity of the $\mathcal{P}$'s context and ensures that $\mathcal{P}$ resumes correctly after interruption.

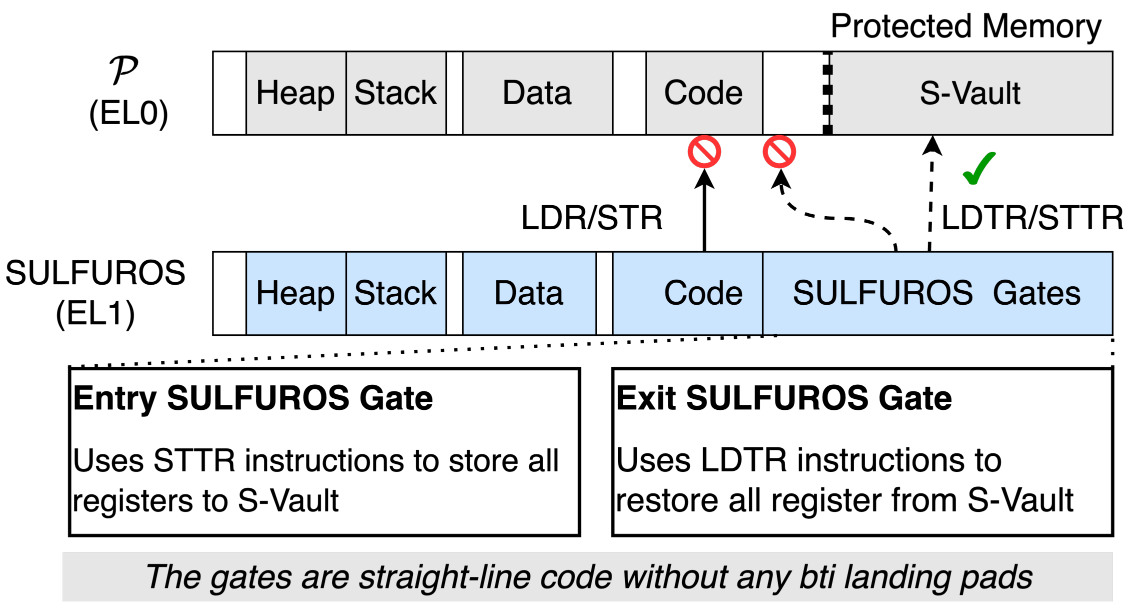

Protected Memory
𝒫
(EL0)
Heap
Stack
Data
Code
S-Vault
LDR/STR
LDTR/STTR
SULFUROS
(EL1)
Heap
Stack
Data
Code
SULFUROS Gates
Entry SULFUROS Gate
Uses STTR instructions to store all registers to S-Vault
Exit SULFUROS Gate
Uses LDTR instructions to restore all register from S-Vault
The gates are straight-line code without any bti landing pads

Figure 3.3: Entry/exit gates in Sulfur save/restore process context in S-Vault using unprivileged loads/stores.

Context Save/Restore Gates in SulfurOS. To protect $\mathcal{P}$'s context (*i.e.*, register state), SULFUROS implements *entry and exit gates* at each entry point and exit point into the kernel (Figure 3.3). Gates are implemented at system call entry/exit points, interrupt handlers, and context-switch code. In addition to storing $\mathcal{P}$'s context on the kernel stack (as normally happens in an OS), the entry gate stores a copy of $\mathcal{P}$'s context in S-Vault, and the exit gate restores $\mathcal{P}$'s context from S-Vault before returning control to the user-space.

Design of SulfurOS Gates. Since S-Vault is in $\mathcal{P}$'s address space, the entry and exit gates must use `LDTR`/`STTR` instructions to access S-Vault. SULFUROS gates are the only places in SULFUROS (aside from user-access functions such as `copy_to_user`) that contain `LDTR`/`STTR` instructions that allow the SULFUROS to write to user-space memory. PAN and SFI ensure that S-Vault is inaccessible from SULFUROS besides the gates and user-access functions (Section 3.3.2).

Normally, accesses to user-space memory via `LDTR`/`STTR` instructions must be SFI-protected to ensure that they do not target arbitrary locations within $\mathcal{P}$'s address space. However, the process context is a data structure of definite size. By ensuring that S-Vault is always allocated at a fixed virtual address in $\mathcal{P}$, the gates hard-code the address of S-Vault and thereby avoid the additional cost of an SFI-check. Control returns from the SULFUROS to $\mathcal{P}$ via exit gates, which use `LDTR` instructions to read the saved process context from S-Vault and restore $\mathcal{P}$'s register state.

```
check_CFA_traced: // Check if app is being traced
  mrs x19, ttbr0_el1 // Load ttbr0_el1 into x19
  and x19, x19, #0x8000000000000000 // Mask all but bit 63
  lsr x19, x19, #0x3f // Shift bit 63 to LSB position
  cmp x19, #0x01 // Compare with 1
  bne skip // Branch to `skip` if not equal
```

Figure 3.4: SulfurOS code that checks whether the process $\mathcal{P}$ (in whose context this code executes) is being attested by CFA.

Entry and Exit Gates in SulfurOS. We present relevant snippets of the low-level AArch64 code in SulfurOS. We start with Figure 3.4, which shows the code that determines whether a particular process $\mathcal{P}$ is traced using CFA. In our prototype, we use the (otherwise unused) bit-63 of the `TTBR0_EL1` register to indicate that $\mathcal{P}$ is being attested via CFA. This bit is set by the Sulfur monitor during process initialization to signal that the operation of $\mathcal{P}$ is being attested by CFA. We fetch the value of `TTBR0_EL1` using the `MRS` instruction, which reads the value of a system register. If the bit is set, the program continues with the execution of SulfurOS gates; otherwise, it skips the gates and proceeds to the kernel logic following the gates.

Figure 3.5 shows relevant code snippets from an entry gate in SulfurOS. The functionality shown in this snippet saves AArch64 general-purpose registers of $\mathcal{P}$ into S-Vault before continuing with kernel execution. This code loads the base address of S-Vault in a reserved register, namely `x19`. The approach of using dedicated registers for user-space CFA has also been explored in prior work on CFA *e.g.,* [7], Chapter 2. In this case, Sulfur requires that the register `x19` be unused in the application $\mathcal{P}$ because its value is overwritten in the entry gate before the values of other registers are saved. In our prototype, we used Blast (Chapter 2) for CFA, which instruments the program $\mathcal{P}$ with CFA-related program instrumentation using LLVM-compiler pass. We simply modify the register allocation policy in this compiler to avoid using `x19` when the binary code of $\mathcal{P}$ is generated. Blast already uses this approach of dedicated registers to save the accumulated path measurements and a pointer to the head of CFA_{Log}, and it was therefore straightforward to extend it to reserve a register to store the head of S-Vault.

Once the base address of S-Vault is loaded in `x19`, all the general-purpose registers of $\mathcal{P}$ are saved at an offset from `x19`. The listing shows all the general-purpose registers (`x0`–`x30`), including `x29` (frame pointer) and `x30` (link register), are saved sequentially using the `STTR` instruction to ensure that the values of the user-space register remain protected during SulfurOS execution.

Finally, Figure 3.6 shows the exit gate, which restores the previously saved general-purpose

```
entry_gate: // Save registers to S-Vault
  ldr x19, =S-Vault_addr // S-Vault base address in x19
  sttr x0, [x19, #8 * 0] // Store register x0
  sttr x1, [x19, #8 * 1] // Store register x1
  ...
  sttr x29, [x19, #8 * 29] // store register x29 (FP)
  sttr x30, [x19, #8 * 30] // Store register x30 (LR)
```

Figure 3.5: Entry gate in SulfurOS: Saving user-space register values to S-Vault during kernel entry.

```
exit_gate: // Restore registers from S-Vault
  ldr x19, =S-Vault_addr // S-Vault base address in x19
  ldtr x0, [x19, #8 * 0] // Restore register x0
  ldtr x1, [x19, #8 * 1] // Restore register x1
  ...
  ldtr x29, [x19, #8 * 29] // Restore register x29 (FP)
  ldtr x30, [x19, #8 * 30] // Restore register x30 (LR)
```

Figure 3.6: Exit gate in SulfurOS: Restoring user-space registers from process context saved in S-Vault.

register state from S-Vault upon exit from kernel. This step ensures that the user-space application resumes with an untampered register state. Like in the entry gate, the exit gate loads the base address of S-Vault into the reserved register `x19`, and the subsequent `LDTR` instructions restore the values of all the registers. We observe that neither the entry nor the exit gates contain any BTI landing pads, thereby preventing any potential control-flow hijacks in SulfurOS from re-executing gate code or use them as part of gadgets.

Other user-space context registers—`SP_EL0` (user stack pointer), `ELR_EL1` (exception return address), and `SPSR_EL1` (saved processor state)—are system registers. They can only be modified using `MSR` (write) instructions from exception level `EL1` or higher. As we discuss in Section 3.4.2.1, all code locations in SulfurOS that should otherwise have `MSR` instructions are replaced with `SMC` instructions that invoke Sulfur monitor. Sulfur monitor in turn restores the values of these registers from S-Vault before returning control back to the process $\mathcal{P}$.

Reliable Execution of SulfurOS Gates. To ensure reliable, secure execution of SulfurOS gates, two key properties must hold: ⓐ gates must not be bypassed on entry/exit to/from the kernel and save/restore the user context correctly, and ⓑ they must execute atomically without being re-executed during normal execution of SulfurOS. Re-execution of the gates can be misused to tamper with the S-Vault. Property ⓐ is ensured by the architecture:

for any exception that arrives from user space (EL0), the AArch64 Linux kernel contains fixed exception vector entries, which follow a standarized register saving prologue before serving the exception. For example, the AArch64 SVC instruction transfers control to the `el0_sync` vector, and on user-mode interrupt entry, control goes to `el0_irq`; these vectors use a common `kernel_entry` code which saves the registers on the kernel stack. SulfurOS injects the entry gate logic inside the `kernel_entry` macro, which executes for all EL0 exceptions. Similarly, the exit gate logic is inserted into the `kernel_exit` macro which restores the register values, placed just before the `ERET` instruction, which is invoked in the `ret_to_user` path to return control to the user space. This placement of gates guarantees that the gates are executed on every transition between user space and kernel, ensuring reliable execution.

SulfurOS enforces property ⓑ using PACBTI-enabled CFI in SulfurOS. PACBTI prevents control-flow hijacks that could allow an attacker to re-enter a gate or jump midway *e.g.,* directly to ERET. Since our implementation of entry/exit gates contains no BTI landing pads, PACBTI ensures they execute exactly once, from start to end, and only through valid control paths. This guarantees that context is reliably saved/restored using gates and that the S-Vault remains protected from tampering via misuse of gates.

3.4.1.3 Protecting Integrity of CFA Measurements in REE

Prior work has used various techniques to trace the execution of program $\mathcal{P}$. Early work on CFA (*e.g.,* [5, 87]) sent all control-flow events to the TEE and maintained a cumulative hash of the trace in the TEE. Subsequent refinements have explored ways to store traces in a more efficient manner in the REE before they are committed to the TEE. Log-based CFA approaches store CFA measurements in an isolated region in the REE (*i.e.,* CFA_{Log}) enabled via SFI ([74, 93], Chapter 2), kernel support [89] or hardware support (*e.g.,* Intel MPK [101]). Some approaches ([7], Chapter 2) record CFA measurements in dedicated registers. Although secure under previous threat models, these approaches are not secure against attacks from SysAdv.

- **Protecting CFA measurements in log-based CFA approaches.** CFA_{Log} is not secure when the REE OS is excluded from the trust boundary. Hence, we must store the CFA_{Log} in a protected region inaccessible to the REE OS *viz.,* S-Vault. As discussed in Section 3.3.2, S-Vault is protected from unauthorized accesses by SulfurOS using PAN load/store instructions within the kernel. In $\mathcal{P}$, only CFA instrumentation can append entries to the CFA_{Log} placed inside S-Vault, it ensured via SFI checks in $\mathcal{P}$ ([74], Chapter 2).

- **Protecting CFA measurements in reserved registers.** CFA approaches that dedicate reserved registers for recording CFA measurements (*e.g.,* [7, 22], Chapter 2) are secure against application-level attacks. However, these registers are accessible to `SysAdv` when they are saved during interrupts and context switches. SULFUR protects the integrity of CFA measurements recorded in dedicated registers by saving the registers in S-Vault during interrupts and context switches. In fact, SULFUR protects *all* saved registers during context switches/interrupts as discussed in Section 3.4.1.2.

3.4.1.4 Protecting Integrity of Attestation Calls

SULFUR facilitates secure communication between $\mathcal{P}$ in the REE and the TEE. CFA instrumentation in $\mathcal{P}$ interacts with the TEE to record CFA measurements and sign the measurements for transmission to the remote verifier. We term each call to the TEE from CFA instrumentation in $\mathcal{P}$ as an *attestation call*. Each attestation call is an `SMC` with specific arguments indicating the TEE functionality to be invoked and any associated parameters. For example, in CFLAT, an `SMC` call will invoke an agent in the TEE to record identity of basic block, and provide this identity as the parameter. $\mathcal{P}$ passes these arguments to SULFUROS via registers; they are stored temporarily on the kernel stack, and subsequently transmitted to the TEE through registers.

To safeguard the integrity of these arguments, we enhance the attestation call mechanism used by $\mathcal{P}$. We establish a shared memory region within S-Vault at a fixed virtual address of $\mathcal{P}$, which is made accessible by both $\mathcal{P}$ and the TEE. We modify the attestation calls in $\mathcal{P}$ to store arguments in this shared region inside S-Vault. Recall that with PAN enabled, SULFUROS cannot modify the contents of S-Vault. However, recognizing that SULFUROS facilitates these calls on behalf of $\mathcal{P}$ and could potentially elide them via runtime attacks, we incorporate an additional verification step within $\mathcal{P}$ itself before making each attestation call. Immediately before initiating an attestation call, $\mathcal{P}$ sets a sentinel bit, `attest-bit`, in the shared memory region in S-Vault. When the agent in the TEE services the attestation call, it resets `attest-bit`. Following an attestation call, instrumentation in $\mathcal{P}$ checks the value of `attest-bit`, which must be 0 if the call was properly serviced. If not, it indicates that SULFUROS has bypassed the attestation call. Note that SULFUROS cannot modify `attest-bit` because it resides in S-Vault.

Lastly, we consider the case wherein SULFUROS attempts to subvert CFA by making additional spurious TEE calls with arbitrary arguments instead of the ones passed by $\mathcal{P}$. CFA measurements are stored in an append-only log in the TEE, and such an attempt by SULFUROS would easily be caught during CFA verification because the log would deviate from the expected execution of $\mathcal{P}$.

3.4.2 System Integrity Guardrails

SulfurOS incorporate guardrails that help build a secure foundation for CFA-centric defenses. These include lifetime integrity protection of the REE system configuration and code of the SulfurOS.

3.4.2.1 System Configuration Protection

SulfurOS is designed so that it cannot directly modify system configuration. In AArch64, system configuration is managed through system registers, accessed via `MRS` (read) and `MSR` (write) instructions [16]. As previously discussed, SulfurOS is implemented by modifying Linux. Each occurrence of `MSR` in Linux is replaced with an `SMC` in SulfurOS, which invokes Sulfur monitor. Upon being invoked, Sulfur monitor updates the system registers, ensuring that updates do not compromise hardware protection mechanisms. Specifically, Sulfur monitor guarantees that WXN, PAN, and MMU protections are never disabled. WXN is enforced by setting the write-execute never (WXN) bit in the system control register (`SCTLR_EL1`) [15]. When `SCTLR_EL1.WXN` is set, writable pages are treated as execute never (XN), enhancing security against code execution from writable memory. MMU functionality is enabled by setting the `SCTLR_EL1.M` bit, crucial for enforcing memory protection (*e.g.,* page permissions). PAN protection is activated by setting the `SCTLR_EL1.PAN` bit, preventing kernel access to user space memory. PAC keys are stored in system registers. By replacing all `MSR` instructions with `SMC` in SulfurOS, we ensure that PAC keys cannot be accessed and tampered with from the REE.

Linux contains several references to `MSR` instructions in exception handling code, which posed a key challenge in making the necessary modifications for SulfurOS. Upon an exception, the processor hands control to privileged software, which must execute a handler specific to that exception. The location of this handler is known as the exception vector. In AArch64, these exception vectors are organized in a sixteen entry table, each entry of which is restricted by the AArch64 specification to be at most 0x80 bytes long [11]. These entries modify various system registers using `MSR`. Attempts to replacing these instructions with `SMC` (with arguments set appropriately) in-place to invoke Sulfur monitor leads to code sequences longer than 0x80 bytes. We created a trampoline for each such exception vector with suitable `SMC` calls to Sulfur monitor, and replaced the `MSR` instructions instead with redirections to this trampoline. This preserved the fixed size requirement of the exception vector while still allowing complete mediation of system register modifications by Sulfur monitor.

3.4.2.2 Protecting Integrity of SulfurOS

Sulfur monitor statically attests the SulfurOS binary to ensure that the correct binary boots in the REE. Additionally, Sulfur monitor verifies and configures the system with hardware protections enabled, guaranteeing correct system configuration upon boot. This ensures the integrity of the booted binary and the system's configuration at boot time. To protect the integrity of SulfurOS' code against runtime attacks from a SysAdv, SulfurOS incorporates:

- **Read-only page tables.** During boot, Sulfur monitor ensures that the code pages are exclusively mapped with read-execute permissions, while data pages are mapped with read-write permissions. It write-protects the page tables of SulfurOS. SulfurOS is thus restricted from modifying its own page table entries. It prevents potential modifications by SysAdv, which could alter SulfurOS' page table permissions to inject new code or enable execution of data pages [24, 49].

- **Double-mapping prevention.** Sulfur monitor maintains a record of mapped physical memory and page permissions in a dedicated data structure to detect double-mapping attacks [24]. In such attacks [24], the same physical page is mapped to multiple virtual addresses with conflicting permissions, *e.g.,* writable in user-space and readable in the kernel. This can allow an attacker to bypass access controls, inject code, or tamper with data that was meant to be protected. In SulfurOS, Sulfur monitor mediates page-table updates, by ensuring compliance with the write-execute-never policy and verifying that no physical page is mapped twice with conflicting permissions.

- **Securely setting page table base register.** SulfurOS does not contain any code that allows direct writes to the translation table base register (`TTBR1_EL1`), which holds the base address of the kernel page table. Sulfur monitor configures `TTBR1_EL1` to reference the correct kernel page table, preventing SulfurOS from using potentially corrupted or duplicate page tables with compromised permissions.

- **User-space code execution prevention.** To prevent SulfurOS from executing user-space code, Sulfur monitor leverages ARM's PXN feature [14]. Sulfur monitor ensures that virtual memory regions belonging to user-space are mapped with PXN permissions. PXN disallows execution of instructions from unprivileged memory when the processor is in privileged mode. Only SulfurOS code pages are given privileged-execute permissions and are mapped as read-only.

- **Control-flow integrity protection of SulfurOS.** Code-injection attacks in SulfurOS are not possible as Sulfur monitor ensures that injected code does not have execute permissions. However, code-reuse within SulfurOS may still be possible. We limit the power of these attacks by enabling PACBTI-based CFI by default in SulfurOS. SulfurOS can also be hardened with more sophisticated control-flow integrity instrumentation [50, 98], though we have not done so in our prototype.

3.5 Security Evaluation

The goal of an adversary is to subvert CFA's path measurement so that the path measurement recorded by the CFA machinery ⓐ is acceptable to a verifier $\mathcal{V}$; and ⓑ deviates from the actual program path taken in $\mathcal{P}$. While prior works have focused on securing CFA in bare-metal settings or atop a trusted REE OS, we evaluate how Sulfur protects CFA when $\mathcal{P}$ executes atop SulfurOS.

We empirically validate Sulfur's effectiveness using a variety of attacks that we launched against a CFA-traced application, Syringe Pump [92] ($\mathcal{P}$, henceforth), a real embedded application used in medical applications. It takes as input a quantity of fluid (bolus), and the direction of syringe movement (dispense/withdraw). For concreteness, we use Blast [95] to CFA-trace $\mathcal{P}$, but our arguments apply to other CFA methods as well. In each case, we demonstrate attacks when $\mathcal{P}$ runs atop Linux as the REE OS, and show that they fail when $\mathcal{P}$ runs atop Sulfur/SulfurOS.

1. **Application code modification/injection.** We carried out a control-flow hijacking attack by modifying code in the `move-syringe` function of $\mathcal{P}$. It implements a loop to administer medicine and the number of loop iterations depend on the input value. Our attack carefully duplicates the code responsible for dispensing medicine inside the loop in the `move-syringe` function while avoiding CFA-instrumentation instructions in $\mathcal{P}$, which in this case appeared at the beginning and end of the loop, recording entry and exit into the loop. Thus, the quantity of dispensed medicine (bolus) is double the requested quantity and this attack does not leave a footprint in CFA_{Log}. To implement this attack, we added malicious code in the REE OS to disable WXN and rewrite $\mathcal{P}$'s binary after it is statically attested at startup.

   This attack does not work when $\mathcal{P}$ executes atop Sulfur. When $\mathcal{P}$ starts, SulfurOS transfers control to Sulfur monitor to perform static attestation of $\mathcal{P}$'s binary which stores the resulting measurement in the TEE. As WXN can be disabled by the REE OS by modifying system registers, SULFUR replaces such instructions in the SULFUROS with calls to SULFUR MONITOR which validates the operation against SULFUR policies and performs it on behalf of the SULFUROS. Hence, SULFUR MONITOR ensures that WXN is always enabled, thus defeating the above attack. Additionally, SULFUR MONITOR write-protects $\mathcal{P}$'s page table and shepherds all changes to it via the TEE. It sets the `TTBR0_EL1` register to reference $\mathcal{P}$'s page table and handles page faults. Collectively, these measures ensure the integrity of $\mathcal{P}$'s code, including CFA instrumentation in $\mathcal{P}$.

2. **CFA state corruption.** We implemented a non-control-data attack to modify the input value to get $\mathcal{P}$ to inject a different quantity of medicine. Further, we corrupted the CFA_{Log} in $\mathcal{P}$ at runtime to hide the attack. For this, we maliciously modified CFA_{Log} (to reflect execution with the unmodified input value) from the REE OS by directly accessing the memory of $\mathcal{P}$ (where CFA_{Log} is stored) when the REE OS receives a request to flush CFA_{Log} to the TEE.

   SULFUR prevents this attack because it protects CFA_{Log} by placing it within the S-Vault region, which is write-protected from SULFUROS via PAN, as indeed is all of $\mathcal{P}$'s memory. PAN cannot be disabled by the OS, as SULFUR prevents any writes to the system registers in the REE. Note that Blast uses a reserved register to store a pointer to the head of CFA_{Log}. When $\mathcal{P}$ cedes control to the kernel, SULFUR stores that register to S-Vault, thus protecting it.

3. **TEE invocation suppression/modification.** To illustrate suppression or modification of TEE invocations from the REE, we modified the REE OS code to suppress/modify TEE calls made by $\mathcal{P}$ to record CFA measurements in the TEE. This attack is best illustrated when CFLAT [5] or OAT [87] are used to instrument $\mathcal{P}$ because they trigger a TEE transition for every control-flow event in $\mathcal{P}$. CFA measurements collected in the REE are committed to the TEE via arguments to a world-switch call. By suppressing TEE calls (or modifying their arguments), we prevented recording in the TEE measurement log malicious control-flow transfers effected in $\mathcal{P}$.

   The attack fails in SULFUR because it ensures that that arguments to the TEE call are passed via S-Vault are protected from any unauthorized modifications from SULFUROS. SULFUR detects attempts to suppress TEE calls to commit CFA state using the `attest-bit` sentinel described in Section 3.4.1.4. In particular, $\mathcal{P}$ detects that the `attest-bit` in S-Vault has not been toggled. Because it can only be toggled by the TEE, it reveals a TEE call suppression attack.

4. **Application execution state modification.** We corrupted the return address (register `x30` in AArch64, saved on the stack) of $\mathcal{P}$ from within the REE OS during a context switch. In particular, by modifying the return address, we skip over the instructions responsible for injecting medicine in the `move-syringe` function. To trigger a context switch, we added call to `sched_yield()` at the loop entry. As a result, each loop iteration caused a context switch. The REE OS at this point modified the return address to skip instructions in the loop. In this case, the CFA instrumentation instructions were only located at the end of the loop body and continue to record the loop execution. Thus, we successfully avoided dispensing medicine without leaving a footprint in the CFA$_{Log}$.

   This attack fails on SULFUR because it saves and restores $\mathcal{P}$'s context to/from the S-Vault using entry and exit gates (Section 3.4.1.2). SULFUROS cannot corrupt the saved context of $\mathcal{P}$ (`x30` is saved to S-Vault). Although SULFUROS can access $\mathcal{P}$'s memory within the confines of PAN using user-access functions (*e.g.,* `copy_to_user()`), SFI checks in their code prevent access to S-Vault. Furthermore, the absence of BTI landing pads in the entry/exit gates ensures that control-flow hijacking attacks within SULFUROS cannot use gate code to access S-Vault.

Formal treatment and Robust CFA. We now provide a formal analysis of the security of user-space CFA atop SULFUR. Let $\mathcal{P}$ be a program with a known control-flow graph (CFG) $G = (V, E)$, where V is the set of basic blocks and $E \subseteq V \times V$ is the set of valid control-flow transitions. For this program $\mathcal{P}$ and a given input $\mathcal{I}$, we define three functions:

- $\pi(\mathcal{P}, \mathcal{I}) = [v_0, v_1, ..., v_n]$, which denotes the *expected execution path* taken by $\mathcal{P}$ on input $\mathcal{I}$, where each $(v_i, v_{i+1}) \in E$. The task of defining this function is that of the verifier $\mathcal{V}$ because this function is used by $\mathcal{V}$ to validate the CFA attestation report collected from the prover platform.

- $\Upsilon(\mathcal{P}, \mathcal{I}) = [u_0, u_1, ..., u_m]$, which is *$\mathcal{P}$'s trace extracted from the CFA attestation report.* That is, it is the control-flow trace recorded by CFA for $\mathcal{P}$ on input $\mathcal{I}$ as $\mathcal{P}$ (suitably compiled with CFA instrumentation) executes atop a platform with SULFUR enabled.

- $\pi_{actual}(\mathcal{P}, \mathcal{I})$, which denotes the *actual execution path* taken by $\mathcal{P}$ on input $\mathcal{I}$ on prover platform. Ideally, the verifier aims to establish that $\pi_{actual}(\mathcal{P}, \mathcal{I}) = \pi(\mathcal{P}, \mathcal{I})$. However, the verifier $\mathcal{V}$ has no way to observe π_{actual} on the prover platform, and must instead use Υ to make various deductions about $\mathcal{P}$'s execution as it processes input $\mathcal{I}$.

We define a predicate $\mathsf{Valid}(\mathcal{P}, \mathcal{I}, \Upsilon, \pi, G)$, used by the $\mathcal{V}$, to check whether a given control-flow trace Υ corresponds exactly to the expected execution path $\pi(\mathcal{P}, \mathcal{I})$.

$$\begin{aligned}\mathsf{Valid}(\mathcal{P}, \mathcal{I}, \Upsilon, \pi, G) \iff & \forall i < |\Upsilon|, (\Upsilon_i, \Upsilon_{i+1}) \in E \\ & \bigwedge \Upsilon(\mathcal{P}, \mathcal{I}) = \pi(\mathcal{P}, \mathcal{I}).\end{aligned}$$

That is, the trace $\Upsilon(\mathcal{P}, \mathcal{I})$ is valid if and only if it exactly matches the expected execution $\pi(\mathcal{P}, \mathcal{I})$ and all transitions in $\mathcal{P}$ are valid CFG edges.

Observe that the Valid predicate accepts $\Upsilon(\mathcal{P}, \mathcal{I})$ to compare against $\pi(\mathcal{P}, \mathcal{I})$, and not $\pi_{actual}(\mathcal{P}, \mathcal{I})$. This is because $\Upsilon(\mathcal{P}, \mathcal{I})$ is the only observable artifact of $\mathcal{P}$'s execution that is available to $\mathcal{V}$. However, in order for $\mathcal{V}$ to make any determinations about $\mathcal{P}$'s execution using $\Upsilon(\mathcal{P}, \mathcal{I})$, it must be an accurate record of $\mathcal{P}$'s execution.

Definition (Robustness) : A CFA mechanism is *robust* if the collected execution trace $\Upsilon(\mathcal{P}, \mathcal{I})$ is an accurate, untampered recording of the actual execution path $\pi_{actual}(\mathcal{P}, \mathcal{I})$ followed by $\mathcal{P}$ as it executes $\mathcal{I}$.

Theorem (Robustness of CFA atop Sulfur) : If the foundational requirements **R1-R4** identified in Section 3.2 hold, then the control-flow trace $\Upsilon(\mathcal{P}, \mathcal{I})$ produced by a CFA-instrumented program $\mathcal{P}$ running on SULFUR system is an accurate, untampered recording of the actual execution path $\pi_{actual}(\mathcal{P}, \mathcal{I})$ of program $\mathcal{P}$ on input $\mathcal{I}$. That is,

$$\mathbf{R1} \wedge \mathbf{R2} \wedge \mathbf{R3} \wedge \mathbf{R4} \implies \Upsilon(\mathcal{P}, \mathcal{I}) = \pi_{actual}(\mathcal{P}, \mathcal{I}).$$

The theorem asserts that if the foundational requirements **R1-R4** hold, SULFUR guarantees that CFA of $\mathcal{P}$ running atop SULFUR records the actual execution of $\mathcal{P}$, and that Υ is not forged by SysAdv, *i.e.,* it enables robust user-space CFA.

Proof Sketch: We show that if **R1-R4** hold, then $\Upsilon(\mathcal{P}, \mathcal{I})$ corresponds to actual execution path $\pi_{actual}(\mathcal{P}, \mathcal{I})$ of $\mathcal{P}$ on input $\mathcal{I}$ in SULFUR system.

- **R1** (WXN) ensures the integrity of the CFA instrumentation throughout the lifetime of $\mathcal{P}$. We used BLAST (Chapter 2) for CFA, which instruments the program $\mathcal{P}$ to record path identifiers, addresses of indirect branches and returns. With WXN, no code injection is possible within $\mathcal{P}$. Furthermore, the careful co-design of SULFUROS (system integrity guardrails) ensures that WXN is enabled for the REE OS kernel as well. All kernel operations in SULFUROS that modify system configurations or memory mappings are shepherded by the TEE (Section 3.4.2.1). Boot-time integrity verification by SULFUR MONITOR ensures that the correct kernel binary is booted. SULFUR MONITOR also configures critical system registers—enabling MMU, DEP, and PAN before booting SULFUROS. It write-protects SULFUROS' code pages, execute-protects its data pages, and write-protects the page-tables of SULFUROS. SULFUROS thus cannot modify its own page access permissions without being intercepted by SULFUR MONITOR. We use PXN to make user-space memory non-executable by the kernel. The power of control-flow hijacking attacks within SULFUROS itself is limited by compiling SULFUROS with PACBTI-enabled CFI. Note that enabling WXN is a pre-requisite for CFI to work correctly [3].

- **R2** ensures secure communication with the TEE. *SMC* calls and their arguments must not be tampered and the TEE must be able to fetch and record CFA information collected in the REE. We ensure this using S-Vault, whose integrity is protected from SULFUROS using PAN. Via the use of PAN, the SULFUROS kernel is prevented from accessing user-space memory. Where access to user-space memory is inevitable in SULFUROS (*e.g.,* user-access functions and gates), it uses `LDTR` and `STTR` instructions to access user-space memory and limits their access to memory regions outside of S-Vault using SFI. Recall also that securely enforcing SFI in the kernel requires that the SFI instrumentation is non-bypassable, a property that we satisfy by ensuring the absence of BTI-landing pads where these user-space memory access instructions are used.

- **R3** requires integrity of execution state during interrupts, and is ensured by saving user-space process context to S-Vault. By the same arguments as above, SULFUROS cannot modify S-Vault.

- Finally, **R4** requires protecting integrity of CFA$_{Log}$. The CFA instrumentation in the user-space program writes the CFA trace $\Upsilon(\mathcal{P}, \mathcal{I})$ to CFA$_{Log}$, which resides in S-Vault. Again, SULFUROS cannot modify CFA$_{Log}$ as a result.

Hence, if **R1-R4** holds, then every CFA trace $\Upsilon(\mathcal{P}, \mathcal{I})$ of $\mathcal{P}$ collected atop SULFUR and sent to $\mathcal{V}$, corresponds to the actual execution $\pi_{actual}(\mathcal{P}, \mathcal{I})$ of $\mathcal{P}$. Thus $\mathcal{V}$ can use $\Upsilon(\mathcal{P}, \mathcal{I})$ as an accurate representation of the actual program execution and check whether $\mathsf{Valid}(\mathcal{P}, \mathcal{I}, \Upsilon, \pi, G)$ holds, which in turn would imply that the actual execution of the program matches the execution expected by $\mathcal{V}$.

In summary, SULFUROS code is co-developed with specific restrictions on its ability to modify user-space state due to the use of PAN load/store instructions. Any code-reuse attacks that work within the constraints imposed by PACBTI-enabled CFI in the kernel must necessarily work under the restrictions imposed by PAN. Therefore, the adversary has a vastly reduced attack surface under which to operate. SULFUR represents—to our knowledge—the first attempt to secure non-bare-metal CFA. Via careful co-design of SULFUROS and the features of the underlying hardware, SULFUR ensures improved security for CFA tracing *any* user-space application running atop a SULFUR-enabled platform.

3.6 Performance Evaluation

We developed our SULFUR prototype on the ARMv8-A base platform Fixed Virtual Platform (FVP), which supports all versions of the Cortex-A architecture [12]. At the time of writing, OP-TEE (which we use for our TEE) does not support the ARMv8.1-A (or above) architecture. Instead, FVP provides a software simulation of an ARMv8.1-A processor in AArch64 mode, enabling us to develop our prototype using ARM's latest features. As a virtual platform, performance measurements on the FVP may not reflect that on real hardware. Therefore, for our performance experiments we use a Raspberry Pi 3 Model B+, which features an ARM Cortex A53 Quad-core 64-bit processor clocked at 1.4GHz and 1GB LPDDR2 SRAM. We use OPTEE in the TEE and run SULFUROS in the REE. As this processor lacks PAN support, we replace the unprivileged load/store (`LDTR`/`STTR`) instructions in SULFUROS with normal load/store (`LDR`/`STR`) instructions for our performance experiments.

Armv8.6-A (and beyond) are only available on high-end devices like smartphones, tablets, and servers, which do not support open development and restrict access to kernel and TEE components. However, our implementation remains architecture-compliant and can support these protections when deployed on hardware with PAN and PACBTI support. The relevant instructions are either replaced or safely ignored in a backward-compatible manner.

For CFA instrumentation of benchmark programs, we use BLAST (Chapter 2). Compared to other methods, Blast reports lower runtime overheads for whole-program CFA because it leverages an optimal algorithm for instrumentation placement and minimizes TEE invocations by batching CFA path measurements. For our experiments, we run the benchmarks atop SULFUROS in the REE. We consider the following application benchmarks, also used in prior work on CFA ([38, 87], Chapter 2), to assess SULFUR's performance impact: (**1**) Embench-IoT [1] is a suite of embedded programs that reflect real embedded systems; (**2**) Syringe Pump [92] is a medical application to administer given doses of medicine; (**3**) Light Controller [61] is an application for remote control of light switches with 'turn-on/off' commands; (**4**) Rover Controller [52] controls the movement of a rover based on commands from a remote operator;

Embench-IOT Programs ↓	Execution Time (s)				
	Baseline-Linux	Sulfur &(Overhead)	CFA-Linux &(Overhead)	CFA-Sulfur &(Overhead)	CFA-Sulfur vs. CFA-Linux
aha-mont64	11.35	11.53 (1.62%)	27.19 (139.66%)	27.76 (144.68%)	2.10%
crc32	11.30	11.49 (1.68%)	17.75 (57.05%)	18.08 (59.97%)	1.86%
cubic	1.72	1.77 (3.11%)	1.86 (8.35%)	1.91 (11.26%)	2.69%
edn	25.41	25.84 (1.69%)	44.40 (74.72%)	45.27 (78.16%)	1.97%
huffbench	13.38	13.62 (1.79%)	24.81 (85.43%)	25.30 (89.11%)	1.99%
matmul-int	32.66	33.21 (1.67%)	42.95 (31.50%)	43.79 (34.07%)	1.96%
minver	4.29	4.37 (1.86%)	7.66 (78.63%)	7.82 (82.28%)	2.04%
nbody	0.51	0.51 (0.66%)	0.72 (42.11%)	0.73 (44.08%)	1.39%
nettle-aes	16.08	16.36 (1.74%)	20.82 (29.46%)	21.23 (32.01%)	1.97%
nettle-sha256	11.77	11.99 (1.87%)	12.71 (7.99%)	12.95 (10.05%)	1.91%
primecount	15.00	15.24 (1.60%)	27.40 (82.69%)	27.69 (84.62 %)	1.06%
sglib-combined	16.41	16.71 (1.81%)	31.93 (94.56%)	32.60 (98.62%)	2.09%
st	0.78	0.79 (1.28%)	1.34 (71.79%)	1.37 (75.64%)	2.24%
tarfind	3.67	3.74 (1.91%)	6.63 (80.56%)	6.76 (84.20%)	2.01%
ud	17.71	18 (1.64%)	29.95 (69.11%)	30.53 (72.41%)	1.95%
Average overhead:		**1.73%**	**63.57%**	**66.74%**	**1.95%**

Table 3.2: Performance of Sulfur on Embench-IoT benchmarks. Number reported are average of 10 runs.

Embedded Applications ↓	Execution Time (s)				
	Baseline-Linux	Sulfur &(Overhead)	CFA-Linux &(Overhead)	CFA-Sulfur &(Overhead)	CFA-Sulfur vs. CFA-Linux
Syringe Pump (500μl)	1.17	1.29 (9.97%)	1.23 (5.13%)	1.35 (15.38%)	9.76%
Syringe Pump (1000μl)	2.35	2.56 (9.09%)	2.41 (2.70%)	2.62 (11.65%)	8.71%
Syringe Pump (2000μl)	4.68	5.12 (9.32%)	4.77 (1.85%)	5.20 (10.96%)	8.94%
Light Controller	2	2.01 (0.50%)	2.06 (3.00%)	2.07 (3.50%)	0.49%
Rover Controller	3.01	3.02 (0.33%)	3.07 (1.99%)	3.07 (1.99%)	0%
MbedTLS-MD5	0.28	0.30 (7.14%)	0.63 (123.81%)	0.65 (132.14%)	3.72%
MbedTLS-SHA256	1.71	1.74 (1.95%)	1.88 (10.14%)	1.92 (12.28%)	1.95%
MbedTLS-RIPEMD160	0.71	0.73 (2.80%)	2.89 (305.14%)	2.94 (312.62%)	1.85%

Table 3.3: Performance of Sulfur on embedded applications. Number reported are average of 10 runs.

(**5**) MbedTLS [2] is an implementation of various cryptographic primitives.

Table 3.2 shows the runtime overhead that Sulfur adds to the Embench-IoT benchmarks. The first column shows the baseline runtime of each application on native Linux-6.7.0. The second column shows the runtime when the uninstrumented benchmark is run atop SulfurOS. We observed an average overhead of 1.73% imposed by SulfurOS.

We then compiled each application with CFA instrumentation, observed the runtime of the CFA-instrumented benchmark atop native Linux, as shown in the third column. This column also shows the overhead compared to the baseline on Linux, with an average overhead of 63.57% observed across all benchmarks. This is the cost of CFA itself and aligns with previous findings ([74], Chapter 2), but is unrelated to this work's contributions. Note that CFA is insecure atop native Linux without the mechanisms implemented in SULFUR. The fourth column shows the raw runtime of CFA-instrumented benchmark atop a SULFUR platform. With SULFUR, these binaries incur a 66.7% overhead over the baseline. Compared to the runtime of CFA-instrumented programs atop Linux, this is an average overhead of just 1.95%.

Table 3.3 shows the performance results for other embedded applications under various configurations. Syringe Pump takes as input the amount of liquid (bolus) to inject. We vary the bolus values and run the program for 500μl, 1000μl and 2000μl bolus amounts. The Syringe Pump application incurs an overhead of roughly 9% overhead on SULFUR compared to native Linux OS. This is because of the large number of system calls that this application makes in its *dispense-medicine* operation, which is implemented as a loop. The program executes over 6800 system calls to inject 500μl liquid, which is doubled for 1000μl and quadrupled for 2000μl, thus causing an overhead of 9%. Other embedded applications such as Light Controller and Rover Controller fare better, incurring only 0.5% and 0.3% overhead, respectively, on SULFUR compared to native Linux OS. These applications make a few hundred system calls.

We then instrumented these applications to perform CFA, and ran the binaries atop a Linux-based REE and a SULFUROS-based REE. CFA-instrumented Syringe Pump incurs 8-9% overhead on SULFUR versus on a Linux-REE. In contrast, Light Controller incurs only 0.49% overhead, and the Rover Controller does not exhibit any noticeable overhead on SULFUR for CFA-instrumented binaries.

Further, our experiments on MbedTLS [2], as shown in Table 3.3, support our findings of low overhead for SULFUR in CFA-instrumented applications compared to the Linux OS. We conducted experiments on hashing algorithms, including MD5, SHA256, and RIPEMD160 with a data size of 5 MB. The overhead incurred by the CFA-instrumented hashing algorithms with SULFUR ranges from 1.8% to 3.7% compared to performance on a Linux REE. Our results indicate that SULFUR does not impose significant overhead on CFA-instrumented applications compared to the Linux OS.

LMBenchmark	**Execution Time (μs)**		
	Linux	**SulfurOS(Overhead)**	
Null	0.89	4.92	(451.75%)
Stat	6.36	10.10	(58.86%)
Fstat	2.26	6.61	(192.14%)
Open/Close	11.41	20.17	(76.68%)
Read	1.44	5.50	(282.17%)
Write	1.31	5.35	(309.15%)
Page fault	2.31	3.13	(35.45%)
Context switch	15.97	33.31	(108.51%)
Signal handler installation	1.52	5.46	(258.87%)
Signal dispatch	12.35	19.33	(56.52%)

Table 3.4: Overhead incurred by Sulfur on system operations measured using LM-Bench micro-benchmarks.

3.6.1 Microbenchmark Evaluation

For an in-depth analysis of the source of overheads in Sulfur, we used the LMBench suite [68], which exercise a variety of system calls in the OS. As Section 3.4 discusses, several operations in SulfurOS operations are shepherded by Sulfur monitor including system configuration, process creation/context switch, memory management and page fault handling. Sulfur monitor saves the context of the SulfurOS and performs the operation after verification. The average round-trip time from SulfurOS to Sulfur monitor is less than 10μsec on our evaluation board.

Table 3.4 presents the runtime overhead imposed by SulfurOS compared to native Linux OS for various LMBench benchmarks. The Null benchmark executes the *getppid()* system call and measures the round-trip time to/from the kernel, while executing minimal code in the kernel. In SulfurOS, however, the code in the entry/exit gates must saves and restores process context. As a result, the raw overhead of the Null benchmark is rather large at about 450%. File operations (Stat. . .Write) fare better, with overheads ranging from 58% to 309% depending on the amount of functionality that the kernel must implement. As majority of the Sulfur checks impact memory mapping verification and correctness of process resumption, we measure overhead on page faults and context switches. For context switching, we measure the latency of switching between six processes. During the context switch, the Sulfur gates save and restore the process's context to/from the S-Vault and the Sulfur monitor sets the value of the TTBR0_EL1 register to reference the process's page tables. During page faults, in addition to process's context switch overhead, it incurs cost of an SMC switch to Sulfur monitor which updates the page table after verification. The overhead incurred by the Page fault benchmark is

Embedded Program ↓	**CFA-Linux Time (s)**	**CFA-Linux % CPU**	**CFA-SULFUR Time (s)**	**CFA-SULFUR % CPU**	**Overhead**
Syringe Pump	2.43	31%	2.64	33%	8.49%
Light Controller	2.10		2.12		1.11%
Syringe Pump	2.45	32%	2.64	34%	7.90%
Rover Controller	3.09		3.12		1.08%
Light Controller	2.08	32%	2.12	33%	1.76%
Rover Controller	3.10		3.11		0.21%
Syringe Pump	2.47	40%	2.68	42%	8.65%
Light Controller	2.12		2.13		0.63%
Rover Controller	3.11		3.14		1.07%

Table 3.5: Performance of Sulfur on concurrent execution of embedded applications. Multiple CFA-traced applications are run simultaneously.

35% while for Context Switch benchmark, it is 108%. Signal handler installation and dispatch incur overheads of 56%-258%.

3.6.2 Scalability Evaluation

We evaluate SULFUR when multiple CFA-traced applications run concurrently. SULFUR ensures that CFA measurements remain interference-free even under simultaneous execution of multiple applications. Each CFA-traced application records its traces within CFA$_{Log}$, which are then securely flushed to TEE storage using a trusted application (TA) in the TEE. Importantly, each CFA-traced application is assigned a unique TA instance in the TEE. This separation ensures that CFA measurements are recorded independently, preventing any potential interference.

Table 3.5 presents the performance results from our scalability evaluation when multiple CFA-traced embedded applications are executed concurrently. We tested three embedded applications, running them in groups of two and three. The results detail the time taken by individual applications as well as the overall peak CPU usage. Our findings show that CPU usage increased by 1-2% when applications ran on SULFUR compared to the Linux REE. The Syringe Pump application, delivering a bolus of 1000μl, experienced an overhead of 8.49% when run alongside the Light Controller, 7.90% when paired with the Rover Controller, and 8.65% when run concurrently with both the Light Controller and Rover Controller. The Light Controller incurred less than 1.76% overhead in all cases, while the Rover Controller maintained an overhead below 1.08% across all configurations. In summary, our evaluation indicate that SULFUR can run multiple CFA-traced applications without imposing significant overhead or compromising the efficiency of CFA.

3.7 Summary of SULFUR

CFA is a useful security technology that allows fine-grained verification of a program's execution on a remote platform. The CFA literature has primarily focused on low-resource embedded platforms where bare-metal execution of programs is the norm. We showed that prior methods cannot directly be applied for CFA when programs execute in a non-bare-metal setting, atop an OS. We then described SULFUR, a system in which a TEE-based monitor (SULFUR MONITOR) and a co-designed OS (SULFUROS) leverage commodity hardware support on AArch64 to enable robust CFA in non-bare-metal settings.

Chapter 4

Related Work

The most closely related work has already been discussed in the Chapter 1. This chapter focuses on additional related work to position this thesis in the broader context of prior research works.

4.1 Remote Attestation

Remote attestation has been explored in various domains [6, 23, 47, 67, 87] to establish the absence of malicious changes to the memory content. Software-based attestation techniques such as SWATT [80] and PIONEER [81] 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 [67]. Moreover, attacks have been proposed on software based attestation [31, 94].

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 [71] 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 [73] 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.

4.2 Control-Flow Attestation

Hardware-based implementations of CFA [43, 44, 100] trace the program's execution path using hardware modules, eliminating the dependence on a TEE. They introduce hardware support to reduce the overheads of CFA on the prover and securely store measurements. LO-FAT [43] and Litehax [44] do not instrument applications, instead implement CFA with stand-alone hardware modules: a branch monitor and a hash engine. Litehax further extends the processor core with custom hardware to compute hash measurements over all executed memory operations, detecting DOP attacks. Atrium [100] enhances LO-FAT and Litehax by securing them against physical attacks on memory. It includes every instruction executed by the prover in the hash generation, storing the hash in an on-chip memory to prevent tampering.

Software-based CFA methods place additional instructions in the program to record its execution. CFLAT [5] is the first work to propose CFA and encodes the paths as a cumulative hash of the basic blocks executed by the program. Several works have optimized the path measurement strategies on the prover platform. OAT [87] attests only certain operations of the application by tracing selective control-flow events rather than all basic block executions. Tiny-CFA [74] proposes CFA for low-end MCU devices, building on top of the formally verified PoX architecture [72]. It stores CFA measurements in a data-memory region protected with SFI. DIAT [6] is an integrity measurement approach that, like BLAST, attempts to address the issue of performance overheads of CFA. 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 [55] 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.

ReCFA [101] proposes an alternative approach to optimizing CFA. 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 [57] 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 CFA 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 [19, 37] require a trap to the kernel to switch memory domains unlike MPK, in which the corresponding operation is done entirely in user-space. ARI [93] generalizes CFA to mission execution integrity and also protects path measurements in REE via SFI.

MG-CFA [54] 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$\times$ 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 CFA. BLAST's overhead, in contrast, is an average of 1.67$\times$ on the Embench-IOT benchmark suite.

ISC-FLAT [69] implements an interrupt-security module in the TEE to protect the program's execution from compromise within ISRs. PEARTS [70] extends Proof of Execution (PoX) to real-time embedded systems. Similar to ISC-FLAT, it protects the execution flow integrity and ensures that interrupts, system calls, and other events must not alter the intended control flow of the code. Additionally, it also provide information about any delays or timing deviations caused by external factors. SpecCFA [32] introduces an application-aware speculative verification approach to reduce the overhead of CFA. By predicting sub-paths of execution based on high-level control-flow knowledge, SpecCFA allows the verifier to speculatively validate traces without matching every edge explicitly, improving audit efficiency without significantly weakening security. SABRE [33] focuses on the verifier's role in runtime attestation. It uses CFA evidence to analyze, debug, and patch binaries, even in the absence of source code and expands the utility of CFA beyond mere validation.

Similar to SULFUR, a few CFA works have targeted non-bare-metal settings. They focus on scaling the CFA methods to more complex applications while trusting the OS. Scarr [89] optimizes the performance of verifier by recording sub-paths instead of a single hash value for entire execution. CFA+ [7, 8] combines CFI with CFA approaches to develop attestation and prevention mechanisms for complex user-space applications. ReCFA [101] uses program analysis to reduce the number of control-flow events recorded at runtime and stores them in an isolated memory region inside the process's address space enabled by Intel MPK [57]. Several works on CFA focus on optimizing the verification of the attestation report. RAGE [38] builds a ML model using program's traces to verify the attestation report provided by the prover. Zekra [42] uses zkSNARKs to enable privacy-preserving execution proofs, outsourcing proof verification to a worker that convinces an untrusted verifier about the verification results without revealing the code of $\mathcal{P}$. These works are complementary to SULFUR, focusing on optimizing CFA in bare-metal settings or using the REE OS as the trust anchor.

4.3 OS-level Protection

Sprobes [49] and Samsung Knox [24] use ARM TrustZone to enforce kernel code integrity by modifying the kernel code to transfer control to TrustZone for introspection of its operations. The design of SULFUR is inspired by Sprobes and Knox. SKEE [25] implements the protections offered by Knox within the kernel itself, eliminating the need for ARM TrustZone. It creates an isolated execution environment within the OS kernel, modifying the kernel to switch to this environment to perform privileged operations (*e.g.,* memory management). Access to the isolated environment is controlled via designated switch gates, ensuring that the CPU can only run either the OS kernel or the isolated execution environment at any given time. Bastion [58] proposes a system call integrity monitor that enforces correct usage of system calls within a program, defending against kernel attacks from userspace (*e.g.,* privilege escalation). TrustShadow [51] protects legacy apps from untrusted OSes by running them in the TEE with a mediating runtime. Virtual Ghost [41] instruments the OS to isolate app memory via a hardware abstraction layer. It enables a secure memory region for applications to store sensitive data which is made inaccessible to the kernel. Ginseng [99] uses a compiler-based approach which modifies the application code to protect the sensitive data. The compiler ensures that sensitive data is always kept in registers and never spilled. At function call sites in the program and kernel entry code, the control is transferred to the TEE which encrypts the sensitive data and stores it in a secure stack to protect its integrity. Other works [28, 84, 86] defend execution of applications using Intel SGX. These works are complementary to our contributions and can be used in conjunction to our defense mechanisms.

Chapter 5

Conclusion and Open Problems

CFA is a powerful security primitive that enables fine-grained verification of program execution on remote platforms. However, achieving practical CFA has been challenging. This thesis presented two key contributions addressing distinct yet fundamental limitations in the CFA landscape.

5.1 Summary of Contributions

First, we demonstrated that whole-program CFA, while desirable for comprehensive attestation, has remained elusive due to the prohibitive runtime overheads of prior approaches. Specifically, existing methods impose an excessive number of TEE domain transitions to record control-flow measurements, rendering them impractical for whole-program paths. To address this, we proposed Blast, a novel CFA approach that combines local logging, optimal instrumentation placement based on Ball-Larus numbering, and software fault isolation. Our evaluation showed that Blast enables scalable whole-program CFA with an average runtime overhead of 67% on embedded benchmarks, significantly outperforming existing techniques.

Second, we explored the applicability of CFA in non-bare-metal settings, such as general-purpose platforms where programs execute atop an operating system. Prior CFA methods assume bare-metal execution and cannot directly handle the complexities introduced by OS abstractions. We presented Sulfur, a system that integrates a TEE-based monitor with a co-designed OS to enable robust CFA for unmodified user-space programs. Sulfur leverages commodity hardware features in AArch64 architectures to protect control-flow integrity and enforce trustworthy measurement collection, thus extending CFA beyond embedded systems to modern computing platforms.

Together, these contributions advance the state-of-the-art in CFA by making whole-program attestation feasible and extending CFA's applicability beyond embedded bare-metal devices. They pave the way for deploying CFA in practical systems, strengthening the foundations for building trustworthy software execution in diverse computing contexts. We now describe future directions for further enhancements in CFA and its applications.

5.2 Potential Avenues for Future Research

Attestation of OS services: User applications depend on the correct execution of OS services (including I/O). For example, the OS provides commands to the medicine dispenser to inject medicine upon request from the corresponding user application. Unreliable execution of OS services can compromise the integrity of operation, with devastating consequences in such applications. CFA approaches proposed thus far, including work on the SULFUR project, have primarily focused on ensuring the integrity of the prover program's execution, and not on ensuring that the operation has been pursued to completion within the OS and hardware. Attestation of the OS's and hardware's actions is a critical yet unexplored area that holds significant promise for improving system security. Building mechanisms for attesting OS services and the actions of hardware peripherals will likely require a novel combination of techniques from the research literature, including control-flow integrity, OS kernel instrumentation, and modern hardware features such as memory domains, support for message-authentication codes, and performance counters.

Attestation for heterogeneous Cyber-Physical Systems (CPS): Attesting heterogeneous CPS presents unique challenges. An example of such a system is a drone, where the companion board processes camera images to generate navigation data — such as "turn left" or "turn right" — which is then sent to another board, the flight controller, for execution. In this scenario, attesting only one component of the system is insufficient; it is crucial to ensure the integrity of the entire pipeline. However, this poses a different set of challenges compared to homogeneous systems, as the software runs across different platforms. The primary challenge lies in stitching together the executions of these distributed software components to assess the overall integrity of the system. This requires developing new verification techniques tailored for heterogeneous environments. Additionally, some components in such environments may lack a TEE. In these cases, revisiting ideas from software attestation could be promising, as the boards are physically connected, making certain techniques more viable.

Trustworthy ML systems: Advancements in ML are rapidly beginning to allow deployment of ML models on edge devices. When such an ML model is deployed on edge devices, similar questions arise as in CFA, albeit in the context of ML models. For example: Was the ML model trained on datasets with specific distributions? Did the embedded system actually execute the ML model in response to the query that was provided in producing the result? Can a proof of execution be provided that does not compromise the secrecy of the ML model itself (since in many cases the ML model is proprietary and cannot be revealed)? With ML models, the traditional notion of "program execution" has changed — from reasoning about paths in an imperative program, to reasoning about how tensors are transformed by layers of a neural network. Evolving program attestation techniques to answer such questions in the context of ML model training and inference forms an important future direction.

Bibliography

[1] Embench^{TM}: A Modern Embedded Benchmark. https://www.embench.org/. 13, 18, 49, 52, 76

[2] Mbed TLS: A C library implementing cryptographic primitives, X.509 certificate manipulation and the SSL/TLS and DTLS protocols. https://github.com/Mbed-TLS. 77, 78

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005. 52, 75

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and Systems Security*, 13(1), 2009. 29

[5] 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 *Proceedings of the ACM Conference on Computer and Communications Security*, 2016. x, 3, 4, 5, 8, 15, 18, 30, 42, 46, 51, 55, 67, 72, 83

[6] 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 *Proceedings of the Networked and Distributed Systems Security Symposium*, 2019. 4, 55, 82, 83

[7] M. Ammar, A. Abdelraoof, and S. Vlasceanu. On bridging the gap between control flow integrity and attestation schemes. In *Proceedings of the 33rd USENIX Security Symposium*, pages 6633–6650, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-44-1. 9, 51, 53, 54, 61, 65, 67, 68, 85

[8] M. Ammar, A. Caulfield, and I. De Oliveira Nunes. Sok: Integrity, attestation, and auditing of program execution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 77–77. IEEE Computer Society, 2024. 85

[9] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the IEEE Symposium on Security and Privacy (Cat. No. 97CB36097)*, pages 65–71. IEEE, 1997. 1

[10] ARM. Security technology building a secure system using TrustZone technology (white paper). *ARM Limited*, 2009. https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2057-00-00-00-00-53-99/PRD29_2D00_GENC_2D00_009492C_5F00_trustzone_5F00_security_5F00_whitepaper.pdf. 2

[11] ARM. AArch64 Exception and Interrupt Handling: Exception vector table, 2025. https://developer.arm.com/documentation/100933/0100/AArch64-exception-vector-table. 69

[12] ARM. Fast Models Fixed Virtual Platforms (FVP) Reference Guide, 2025. https://developer.arm.com/documentation/100966/1126. 76

[13] ARM. Privileged access never, 2025. https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Registers/PAN--Privileged-Access-Never. 52, 57, 59

[14] ARM. ARMv8-M Memory Model and Memory Protection User Guide: Significance of XN and PXN bits, 2025. https://developer.arm.com/documentation/107565/0101/Memory-protection/Significance-of-XN-and-PXN-bits. 52, 57, 70

[15] ARM. Cortex-A Series Programmer's Guide for ARMv8-A: The system control register, 2025. https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/System-registers/The-system-control-register. 69

[16] ARM. Cortex-A Series Programmer's Guide for ARMv8-A: ARM System Registers, 2025. https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/System-registers. 69

[17] ARM. Trusted Firmware-A Documentation, 2025. https://trustedfirmware-a.readthedocs.io/en/latest/. 62

[18] ARM. Security technology, 2025. https://developer.arm.com/documentation/PRD29-GENC-009492/c. 1, 2

[19] ARM. Memory domains (page b3-31), 2025. ARM v7A/v7R Architecture Reference Manual. 84

[20] ARM. Using pac and bti features in applications, 2025. https://developer.arm.com/documentation/109576/0100/Using-PAC-and-BTI-features-in-applications. 55, 57, 58

[21] ARM. Smc calling convention (smccc), 2025. https://developer.arm.com/documentation/den0028/a/. 62

[22] Md Armanuzzaman, E. Kirda, and Z. Zhao. Enola: Efficient control-flow attestation for embedded systems. *arXiv preprint arXiv:2501.11207*, 2025. 68

[23] N. Asokan, F. Brasser, A. Ibrahim, A-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann. Seda: Scalable embedded device attestation. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2015. 82

[24] 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 *Proceedings of the ACM Conference on Computer and Communications Security*, 2014. 9, 33, 34, 51, 52, 55, 58, 70, 85

[25] A. M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning. SKEE: A lightweight Secure Kernel-level Execution Environment for ARM. In *Proceedings of the Network and Distributed System Security Symposium*, 2016. 85

[26] T. Ball and J. Larus. Efficient path profiling. In *Proceedings of the ACM/IEEE Symposium on Microarchitecture*, Dec 1996. 13, 19, 21, 23, 24

[27] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2003. 52, 58

[28] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. *ACM Transactions on Computer Systems*, 33(3), September 2015. 85

[29] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, 2011. 4, 16

[30] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2008. 33

[31] 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. 82

[32] A. Caulfield, L. Tyler, and I. D. O. Nunes. SpecCFA: Enhancing Control Flow Attestation/Auditing via Application-Aware Sub-Path Speculation . In *Proceedings of the Annual Computer Security Applications Conference*, pages 563–578, Los Alamitos, CA, USA, December 2024. IEEE Computer Society. doi: 10.1109/ACSAC63791.2024.00055. 84

[33] A. I. Caulfield, N. Rattanavipanon, and I. D. O. Nunes. Run-time attestation and auditing: The verifier's perspective. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec 2025, page 16–27, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715303. doi: 10.1145/3734477.3734710. 84

[34] S. Checkoway, L. Davi, A. Dmitrienko, A-R. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming Without Returns. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2010. 4, 16

[35] L. Chen, R. Landfermann, H. Löhr, M. Rohe, A-R. Sadeghi, and C. Stüble. A protocol for property-based attestation. In *Proceedings of the ACM workshop on Scalable trusted computing*, pages 7–16, 2006. 1

[36] L. Chen, H. Löhr, M. Manulis, and A-R. Sadeghi. Property-based attestation without a trusted third party. In *Proceedings of the International Conference on Information Security*, pages 31–46. Springer, 2008. 1

[37] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. Shreds: Fine-grained execution units with private memory. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2016. 84

[38] M. Chilese, R. Mitev, M. Orenbach, R. Thorburn, A. Atamli, and A-R. Sadeghi. One for all and all for one: GNN-based control-flow attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2024. 3, 76, 85

[39] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer. Protecting bare-metal embedded systems with privilege overlays. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2017. 15

[40] 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 International Symposium on Research in Attacks, Intrusions and Defenses*, 2022. 33

[41] J. Criswell, N. Dautenhahn, and V. Adve. Virtual Ghost: Protecting applications from hostile operating systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, page 81–96, 2014. 85

[42] H. B. Debes, E. Dushku, T. Giannetsos, and A. Marandi. Zekra: Zero-knowledge control-flow attestation. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, page 357–371, 2023. 85

[43] 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 ACM/EDAC/IEEE Design Automation Conference*, 2017. 83

[44] G. Dessouky, T. Abera, A. Ibrahim, and A-R. Sadeghi. LiteHAX: lightweight hardware-assisted attestation of program execution. In *Proceedings of the International Conference on Computer Aided Design*, 2018. 4, 8, 83

[45] R. Doyle, R. Some, W. Powell, G. Mounce, M. Goforth, S. Horan, and M. Lowry. High performance spaceflight computing (hpsc) next-generation space processor (ngsp): a joint investment of nasa and afrl. In *Proceedings of the Workshop on Spacecraft Flight Software*, pages 1–19, 2013. 9, 51

[46] R. Felker et al. Musl libc: An implementation of the standard library for linux-based systems, 2025. `https://musl.libc.org/`. 39

[47] T. Fraser, J. Molina, and W. Arbaugh. Copilot–a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the USENIX Security Symposium*, 2004. 82

[48] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proceedings of the National Computer Security Conference*, pages 305–319, 1989. 1

[49] X. Ge, H. Vijayakumar, and T. Jaeger. SPROBES: Enforcing Kernel Code Integrity on the TrustZone. In *Proceedings of the IEEE Workshop on Mobile Security Technologies*, 2014. 33, 34, 55, 58, 70, 85

[50] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *Proceedings of the IEEE European Symposium on Security and Privacy*, 2016. 71

[51] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services*, page 488–501, 2017. ISBN 9781450349284. 85

[52] Gwaltrip. Github. rover controller, 2025. `https://github.com/Gwaltrip/RoverPi/tree/master/tcpRover`. 76

[53] 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 *Proceedings of the IEEE Symposium on Security and Privacy*, 2016. 4, 16

[54] 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 *Proceedings of the IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom/BigDataSE)*. IEEE, 2019. 8, 84

[55] 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 *Proceedings of the IEEE International Conference on High Performance Computing and Communications*, 2020. 4, 83

[56] Intel. Intel SGX for linux. `https://github.com/intel/linux-sgx`. 1

[57] Intel. Intel-64 and IA-32 architectures software developer's manual, 2018. `https://software.intel.com/en-us/articles/intel-sdm`. 51, 52, 84, 85

[58] C. Jelesnianski, M. Ismail, Y. Jang, D. Williams, and C. Min. Protect the system call, protect (most of) the world with bastion. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 528–541, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399180. doi: 10.1145/3582016.3582066. 85

[59] A. Joseph, N. Yadav, V. Ganapathy, and D. Behl. A contributory public-event recording and querying system. In *Proceedings of the IEEE/ACM Symposium on Edge Computing*, pages 1–14, Los Alamitos, CA, USA, dec 2023. IEEE Computer Society. doi: 10.1145/3583740.3628445. vi

[60] A. Joseph, N. Yadav, V. Ganapathy, D. Behl, and P. Jayachandran. Data protection in permissioned blockchains using privilege separation. In *Proceedings of the International Conference on Communication Systems & Networks*, pages 748–756. IEEE, 2023. vi

[61] J. Judin. Github. light controller, 2025. `https://github.com/Barro/light-controller`. 76

[62] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight kernel protection against Return-to-User attacks. In *Proceedings of the USENIX Security Symposium*, pages 459–474, Bellevue, WA, August 2012. ISBN 978-931971-95-9. 9, 51

[63] 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 *Proceedings of the IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 115–124. IEEE, 2009. 1

[64] 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 *Proceedings of the Networked and Distributed Systems Security Symposium*, 2018. 15

[65] D. Koisser, R. Mitev, N. Yadav, F. Vollmer, and A-R. Sadeghi. Orbital trust and privacy: SoK on PKI and location privacy challenges in space networks. In *Proceedings of the USENIX Security Symposium*, pages 6093–6111, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-44-1. vi

[66] J. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, May 1999. 13, 19, 24, 26, 30, 31

[67] Y. Li, J. M. McCune, and A. Perrig. Viper: Verifying the integrity of peripherals' firmware. In *Proceedings of the ACM conference on Computer and communications security*, pages 3–16, 2011. 82

[68] L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 1996. 79

[69] A. J. Neto and I. D. O. Nunes. ISC-FLAT: On the Conflict Between Control Flow Attestation and Real-Time Operations. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2023. 33, 53, 54, 84

[70] A. J. Neto, N. Rattanavipanon, and I. D. O. Nunes. PEARTS: Provable Execution in Real-Time Embedded Systems . In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 3765–3782, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. doi: 10.1109/SP61157.2025.00047. 84

[71] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik. Vrased: A verified hardware/software co-design for remote attestation. In *Proceedings of the USENIX Security Symposium*, pages 1429–1446, 2019. 82

[72] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik. Apex: a verified architecture for proofs of execution on remote devices under full software compromise. In *Proceedings of the USENIX Security Symposium*, USA, 2020. USENIX Association. ISBN 978-1-939133-17-5. 83

[73] I. D. O. Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik. On the TOCTOU problem in remote attestation. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2021. 82

[74] I. D. O. Nunes, S. Jakkamsetti, and G. Tsudik. Tiny-CFA: Minimalistic control-flow attestation using verified proofs of execution. In *Proceedings of the Design, Automation and Test Conference*, 2021. 4, 8, 54, 55, 61, 67, 78, 83

[75] A. Papoulis. *Bernoulli Trials.* Probability, Random Variables, and Stochastic Processes (2nd ed.). McGraw-Hill, 1984. 47

[76] OP-TEE Project. OP-TEE: Open portable trusted execution environment. `https://www.op-tee.org/`. 15, 29, 32, 62

[77] B. Roberts. Enabling PAC and BTI on AArch64 for Linux, November 2024. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enabling-pac-and-bti-on-aarch64. 52

[78] A-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *Proceedings of the workshop on New security paradigms*, pages 67–77, 2004. 1

[79] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the USENIX Security Symposium*, 2004. 1

[80] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2004. 82

[81] 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 *Proceedings of the ACM Symposium on Operating Systems Principles*, 2005. 82

[82] J. Seward. bzip2, 2025. Linux man page. https://linux.die.net/man/1/bzip2. 41

[83] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM conference on Computer and communications security*, pages 552–561, 2007. 4, 16, 38

[84] K. Shanker, A. Joseph, and V. Ganapathy. An evaluation of methods to port legacy code to sgx enclaves. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1077–1088, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409726. URL https://doi.org/10.1145/3368089.3409726. 85

[85] P. Sheng, N. Yadav, V. Sevani, A. Babu, SVR Anand, H. Tyagi, and P. Viswanath. Proof of backhaul: Trustfree measurement of broadband bandwidth. In *Proceedings of the Network and Distributed System Security Symposium*, 2024. vi

[86] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux applications with SGX enclaves. In *Proceedings of the Networked and Distributed Systems Security Symposium*, 2017. 85

[87] Z. Sun, B. Feng, L. Lu, and S. Jha. OAT: Attesting operation integrity of embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2020. x, 1, 3, 4, 5, 15, 18, 34, 46, 51, 53, 55, 56, 67, 72, 76, 82, 83

[88] G. Tan. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends® in Privacy and Security*, 1(3):137–198, 2017. 60

[89] F. Toffalini, E. Losiouk, A. Biondo, J. Zhou, and M. Conti. Scarr: Scalable runtime remote attestation for complex systems. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*, 2019. 4, 8, 9, 51, 53, 54, 59, 67, 85

[90] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 1993. 13, 29, 51, 60

[91] M. Walfish and A. Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2), February 2015. 3

[92] T. Walker. Github. open syringe pump, 2025. https://github.com/manimino/OpenSyringePump. 41, 71, 76

[93] J. Wang, Y. Wang, A. Li, Y. Xiao, R. Zhang, W. Lou, Y. T. Hou, and N. Zhang. ARI: Attestation of real-time mission execution integrity. In *Proceedings of the USENIX Security Symposium*, pages 2761–2778, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. 53, 54, 56, 59, 67, 84

[94] G. Wurster, P. C. Van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'05)*, pages 127–138. IEEE, 2005. 82

[95] N. Yadav and V. Ganapathy. Whole-program control-flow path attestation. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 2680–2694, 2023. v, 71

[96] N. Yadav, F. Vollmer, A-R. Sadeghi, G. Smaragdakis, and A. Voulimeneas. Orbital shield: Rethinking satellite security in the commercial off-the-shelf era. In *Proceedings of the Security for Space Systems Conference*, pages 1–11. IEEE, 2024. vi

[97] N. Yadav, H. Salunke, D. Tejas, and V. Ganapathy. Non-bare-metal userspace control-flow attestation. In *Proceedings of the Annual Computer Security Applications Conference*, 2025. v

[98] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim. In-Kernel Control-Flow integrity on commodity OSes using ARM pointer authentication. In *Proceedings of the USENIX Security Symposium*, pages 89–106, Boston, MA, August 2022. ISBN 978-1-939133-31-1. 71

[99] M. H. Yun and L. Zhong. Ginseng: Keeping Secrets in Registers When You Distrust the Operating System. In *Proceedings of the Network and Distributed System Security Symposium*, 2019. 85

[100] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A-R. Sadeghi. ATRIUM: Runtime attestation resilient under memory attacks. In *Proceedings of the International Conference on Computer Aided Design*, 2017. 83

[101] Y. Zhang, X. Liu, C. Sun, D. Zeng, G. Tan, X. Kan, and S. Ma. ReCFA: Resilient Control-Flow Attestation. In *Proceedings of the Annual Computer Security Applications Conference*, 2021. 4, 9, 51, 53, 54, 55, 67, 83, 85