Panoply: Low-TCB Linux Applications with SGX Enclaves

Shweta Shinde
National University of Singapore
shweta24@comp.nus.edu.sg

Dat Le Tien[†]
University of Oslo
dattl@ifi.uio.no

Shruti Tople
National University of Singapore
shruti90@comp.nus.edu.sg

Prateek Saxena
National University of Singapore
prateeks@comp.nus.edu.sg

***Abstract*—Intel SGX, a new security capability in emerging CPUs, allows user-level application code to execute in hardware-isolated enclaves. Enclave memory is isolated from all other software on the system, even from the privileged OS or hypervisor. While being a promising hardware-rooted building block, enclaves have severely limited capabilities, such as no native access to system calls and standard OS abstractions. These OS abstractions are used ubiquitously in real-world applications.**

In this paper, we present a new system called Panoply which bridges the gap between the SGX-native abstractions and the standard OS abstractions which feature-rich, commodity Linux applications require. Panoply provides a new abstraction called a *micro-container* (or a "micron"), which is a unit of code and data isolated in SGX enclaves. Microns expose the standard POSIX abstractions to application logic, including access to filesystems, network, multi-threading, multi-processing and thread synchronization primitives. Further, Panoply enforces a strong integrity property for the inter-enclave interactions, ensuring that the execution of the application follows the legitimate control and data-flow even if the OS misbehaves. Thus, commodity Linux applications can enhance security by splitting their application logic in one or more microns, or by importing micron-libraries, with little effort. In contrast to previous systems that enable comparable richness, Panoply offers two orders of magnitude lower TCB (about 20 KLOC in total), more than half of which is boiler-plate and can be automatically verified in the future. We demonstrate how Panoply enables much stronger security in 4 real-world applications — including Tor, OpenSSL, and web services — which can base security on hardware-root of trust.

I. Introduction

Privilege separation and isolation are cornerstones in design of secure computer systems. Machine isolation is used for designing fault-tolerant network services, virtualization for isolating OSes, library OSes and containers for isolating applications. However, these primitives trust a privileged software component (e.g. a hypervisor or OS) for ensuring their claimed security guarantees. For several decades, malware has been a threat to privileged software layer, often targeting vulnerabilities in privileged code such as the OS. In this paper, we envision providing the benefits of privilege separation and isolation based on a strong line of defense against OS-resident malware. Such a defense is based on a new trusted computing primitive, which can isolate a sensitive user-level application from a compromised OS. Hardware support for this primitive has become available in commodity CPUs in the form of Intel SGX, which can run such hardware-isolated application instances in *enclaves* [40]. Intel SGX provides a hardware-isolated memory region which can be remotely attested. SGX hardware supports execution of many enclaves simultaneously.

Recent research has demonstrated how to enforce useful low-level guarantees using SGX — for instance, protection of certain cryptographic keys in memory [30], [32], [42], [44], [47], verifiable execution of code snippets [51], and authenticated data delivery [56]. While these properties are useful, their applicability has been limited to small, selected pieces of application logic rather than end-to-end applications. For larger, richer applications, the best known approache has been to use library OSes [21], [52]. In such architectures, the application is bundled together with a large TCB of millions of lines of code, emulating the OS logic inside the enclave.

In this paper, we propose a new system called Panoply, designed with an eye towards minimizing TCB and yet offering rich OS abstractions to enclaved code. Panoply introduces a new abstraction we call a *micro-container* (or "micron" for short). A micron is a unit of application logic which runs on the Intel SGX hardware enclaves — thus it offers a strong isolation against an adversarial OS. Microns expose the rich gamut of standard Linux abstractions to application logic, much more expressiveness in enclave-bound code than several previous systems (e.g. Haven). For instance, micron-enabled logic can readily use multi-processing (`fork-exec`), multi-threading, event registration / callbacks (e.g. signals), in addition to supporting the standard Linux system calls.

Panoply prioritizes a minimal TCB over performance as a goal. It uses a simple design philosophy of *delegate-rather-than-emulate* that contrasts previous systems (e.g. library OSes). Panoply delegates the implementation of OS abstractions to the underlying OS itself, rather than emulating it inside the enclave. Panoply microns implement a small set of checks which enables them to detect malicious response from the OS, and abort if so. In line with this design choice, unlike library OSes, Panoply does not "virtualize" microns by giving them each their own namespace. This choice eliminates a massive amount of namespace management logic

[†]This work was done while the author was a visiting graduate intern at National University of Singapore.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-1891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23500

that emulates the underlying OS from the enclave TCB. With these simple design principles, we show that the total TCB of PANOPLY can be about 20 KLOC (apart from the original application logic), which is 2 orders of magnitude smaller than previous LibraryOS systems. We believe such TCB is within realm of automated verification in the near future.

A second feature of PANOPLY is that it enables a plug-and-play architecture, wherein security architects can create as many microns as needed and host them across multiple OS processes. Such a design keeps compatibility with multi-process and multi-threaded application designs — several applications such as servers inherently use multiple processes for security (e.g. for privilege separation), performance, as well as for isolating crash failures (better availability). Security architects can split monolithic applications across multiple microns easily, or import security-sensitive libraries that are implemented as microns. The PANOPLY architecture allows analysts to partition application by adding modest amount of annotation to source code and compiling with PANOPLY's infrastructure. PANOPLY instruments each micron to ensure that all inter-micron control and data-flow interactions are secured against the OS. In effect, PANOPLY ensures that an application partitioned into multiple microns will either execute with the same control and data-flow as the unpartitioned original application, even under adversarial influence, or abort. To achieve such a guarantee, we propose defenses that extend beyond simple data tampering attacks (e.g. Iago attacks [26]) — our defenses enforce control and data-flow integrity for inter-enclave transitions, with freshness and authentication guarantees built-in.

PANOPLY microns expose expressive OS abstractions to application logic. To support these, we make several conceptual advances in its design that are not offered by off-the-shelf Intel SDKs for SGX [8]. First, microns expose the POSIX abstraction of creating threads on demand, dynamically requesting as many threads as the application needs. Previous systems have limited this design to a pre-determined number of static threads executable in an enclave. PANOPLY runtime library multiplexes a dynamic number of threads over multiple underlying enclaves. Second, PANOPLY proposes several designs for supporting the semantics of `fork-exec`. PANOPLY allows microns to be hosted across multiple OS processes. Third, PANOPLY supports the `pthreads` synchronization interfaces, which includes mutexes, barriers, and so on using SGX-specific abstractions. Finally, as explained earlier, PANOPLY automatically embeds checks at the micron interfaces, ensuring that inter-enclave control and data-flow conforms to that of the original code.

Results. We show 4 case studies of real applications that use expressive features. The first case study is Tor, a popular distributed anonymous communication service [11]. We show how Tor can leverage PANOPLY micron to provide a strong security for its directory service protocol, basing security on the SGX hardware-root-of-trust. The second case study is on a web-server called H2O, which can self-attest the correctness / integrity of the served content (e.g., such as in CDNs, serving static content over HTTPS). We also support two case studies of popular libraries, OpenSSL and FreeTDS, that can be imported as microns in other host application. The libraries can be secured to protect secret keys and provide higher-level protocol guarantees for their host application beyond key-protection. In all of our case studies, the porting effort is modest, incurring average 905 lines of code changes.

We have compared PANOPLY application to a state-of-the-art Linux library-OS called Graphene-SGX [3], [52] that supports SGX hardware. First, PANOPLY applications have two order of magnitude smaller TCB. Second, we find that the performance of the two on our case studies is comparable. We provide a detailed breakdown of the performance overheads on real hardware in both systems. We find that most source of overhead is from the Intel's SDK, incurred for creating and initializing empty enclaves. PANOPLY-specific code introduces an additional average CPU overhead of 24%.

Contributions. In summary, we make the following contributions in this paper:

- PANOPLY *System.* PANOPLY is the first system which supports applications with multi-threading, multi-processing, event management in enclaves. Our inter-micron flow integrity ensures that the applications preserve the high-level guarantees.
- *Usage in real-world Applications.* We retrofit 4 applications into PANOPLY architecture that require on an average 905 lines of code changes.
- *Evaluation.* After porting to PANOPLY, we report a performance overhead of 24% and an average TCB increase of 19.62 KLOC per application. In comparison to previous systems, PANOPLY reduces the TCB by 2 orders of magnitude while lowering the performance overhead by $5 - 10\%$.

II. PROBLEM

A. Background: Intel SGX Enclaves

Existing hardware-based / hypervisor-based defenses against compromised OSes provide an isolated execution space for executing user-level applications. In the case of SGX, these are referred to as enclaves, and a single application process can comprise of one or more enclaves. SGX assures the confidentiality and integrity of all the sensitive code and data contained within an enclave. The Intel SGX SDK provides a function call mechanism for SGX applications via `ECALL` and `OCALL`. Specifically, an `ECALL` is a trusted function call that enters an enclave and `OCALL` is an untrusted function call that leaves an enclave [17]. Thus, a user-level application can invoke code inside an enclave via an Enclave Call (`ECALL`) and get the return values. The enclave can invoke an Outer Call (`OCALL`) to execute a function in the untrusted portion of the application and receive a return value. The enclave code can access all the application memory outside whereas, the non-enclave code cannot access the enclave's memory. SGX CPU supports local and remote attestation, so as to check if the enclave has loaded the correct code. To this end, the CPU computes a measurement by securely hashing the enclaves initial state. Thus an entity can attest the measurement to verify the initial state of the enclave confirming that the initial state is "clean" [18], [23]. The enclave is created and loaded as expected without OS tampering.

```
 1 session_t session;
 2 certificate_credentials_t xcred;
 3 /* Specify the callback function to be used*/
 4 #begin privilege_enclave
 5  certificate_set_verify_function (xcred, _callback);
 6 #end privilege_enclave
 7 /* Initialize TLS session */
 8 init (&session, TLS_CLIENT);
 9 /* Set non-default priorities */
10 if(non-default)
11  #begin privilege_enclave
12   priority_set_direct (session, "%UNSAFE_RENEGO");
13  #end privilege_enclave
14 err = handshake(session);
15 ... }
16
17 static int _callback (session_t session) {
18   x509_crt_t cert;
19   const char *hostname;
20   ...
21   #begin privilege_enclave
22    ret = x509_crt_check_hostname (cert, hostname);
23   #end privilege_enclave
24   if (!ret)
25     return CERTIFICATE_ERROR;
26   ...
27   /* Validation successful, continue handshake */
28   return 0;
29 }
30
31 static SSL_CTX *tds_init_ssl(void)
32 {...
33     tds_mutex_lock(&tls_mutex);
34     if (!tls_initialized) {
35       SSL_library_init();
36       tls_initialized = 1;
37     }
38     tds_mutex_unlock(&tls_mutex);
39 ...}
```

Fig. 1. Code snippet from FreeTDS application for certificate validating of remote database server's certificate using OpenSSL. The #pragmas are added while porting the application to PANOPLY.

B. Attacks & Challenges

To enable end-to-end security guarantees in real applications, we aim to address three main challenges: (1) support for rich OS abstractions, (2) secure interactions between multiple enclaves, and (3) a minimal additional TCB. We demonstrate the gap in existing abstractions to meet these challenges with an example, for ensuring higher-level security properties.

Example. Consider FreeTDS [2], an application that implements a streaming protocol (TDS [9]) for remote databases. Figure 1 shows a FreeTDS code snippet, which shows that the application uses the OpenSSL SSL/TLS library to establish a secure channel with a remote database. Each time it connects to a remote database, FreeTDS checks if the remote database is an authorized service and has a valid SSL certificate, by connecting to trusted certificate manager service. Our goal is to fortify the FreeTDS application against a compromised OS, ensuring a key end-to-end security property — the FreeTDS application accepts a certificate as valid if and only if the original application would have accepted it as is valid. The assumption is that the original application is bug-free, but the system administrators wish to secure it against the hosting service provider's infrastructure, which may be infected. We can achieve such a defense by using Intel SGX enclaves.

In this example, we wish to run the FreeTDS application and the trusted certificate manager, each in their own separate

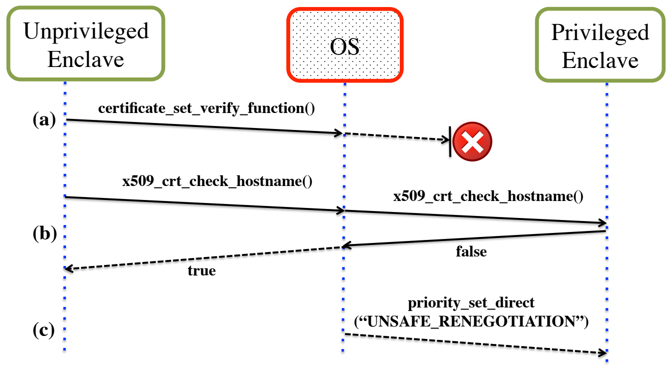

Fig. 2. (a) Call dropping, b) call spoofing and c) call replay attacks perpetrated by the OS during inter-enclave interactions.

SGX enclaves. The trusted certificate manager service is shared across many applications, and by principle of least privilege, is the only service with access to the "root certificate". As can be seen in Figure 1, the FreeTDS application validates a presented certificate by invoking the trusted certificate manager via standard OpenSSL interfaces. Specifically, the FreeTDS application does this in 3 steps: it registers a callback with the trusted certificate manager (Line 5), passes configuration options (Line 12), and invokes the certificate check (Line 22).

Attacks on Inter-Enclave Interactions. Note that the FreeTDS application enclave invokes the certificate manager enclave several times, passing rich data. Such communication necessarily goes through an adversarial channel under the OS control (e.g. an IPC call or control transfer in unprotected code). The OS knows the standard OpenSSL interfaces and the semantics of its interfaces. If enclaves use the standard interfaces of the Intel SDK, the OS can subvert the application's guarantee by causing FreeTDS to accept invalid certificates. We show concrete attacks on this interface in Figure 2. First, the OS can drop the call on Line 5, thereby disallowing the application to register a callback. The attack is powerful because abort fails *silently*, with the effect that the certificate validity checks on Line 22 never execute. A second attack opportunity is to effect a session downgrade attack, by forcing the certificate manager to re-negotiate weak SSL parameters [12], [13]. The SSL protocol has a known flaw called a session re-negotiation vulnerability, which is patched by the call on Line 12. However, the OS can abort this message and this causes the certificate validation to proceed with unsafe defaults silently. As a third example, the OS can perform a session downgrade by *replaying* a call from a different (previous) session. Specifically, the OS can record the inter-enclave message transcript from a different session consisting of a invalid certificate with weaker parameters (requesting `UNSAFE_RENEGOTIATION`). The previous session would have failed; however, the OS can replay one recorded message from that transcript in a session with a strong certificate to cause it to downgrade. A final and fourth attack is on Line 22. This is a data replay attack wherein the certificate manager returns `false` (signaling an invalid certificate), but the OS drops the message and replays a `true` return value from a previous execution run.

These attacks highlight that applications that aren't designed with the objective of running on enclaved-abstraction

TABLE I. COMPARISON OF PROGRAMMING CONSTRUCT SUPPORT AND SECURITY PROPERTIES OF PANOPLY AND OTHER EXISTING SYSTEMS

Abstr actions	Sys Calls	Multi Threading		Thread Synch		Fork Exec	STI	Low TCB
		Stat	Dyn	Mutex	All			
Intel SDK	✗	✓	✗	✓	✗	✗	✗	✓
Haven	✓	✓	✓	✓	✓	✗	✗	✗
SGX Graphene	✓	✓	✓	✓	✓	✓	✗	✗
PANOPLY	✓	✓	✓	✓	✓	✓	✓	✓

will be susceptible to subtle vulnerabilities. Further, there is gap between the SGX-native low-level guarantees (of remote attestation and memory isolation) and those needed to ensure safe end-to-end execution of applications. Several previous works have enforced confidentiality of private keys and authenticated data delivery, while our emphasis is on securing the end-to-end application semantics.

Supporting Rich OS Abstractions With Low TCB. The example highlights an even more basic challenge in supporting real-world applications. The FreeTDS code snippet makes extensive use of OS abstractions. The code is multi-threaded, where a new thread of execution is dynamically created for each request to initiate connection with a new server. Line $33-38$ in Figure 1 shows that all threads use mutex objects for synchronizing operations such as initializing the TLS library. The original application is not designed to be executed in multiple enclaves, which do not share any address space (unlike threads). The application assumes it can create an arbitrary number of threads at runtime on-demand. FreeTDS uses standard system calls, such as `gethostname()` and `poll ()` (implicitly for callback registrations). However, SGX and Intel SDK *do not* provide native support for any of these abstractions. Table I shows the gap between the expressiveness offered by the Intel SDK, library OSes and the requirements of real-world applications. The research question is how to enable support for such abstractions with minimal effort, enabling security architects to quickly experiment with ways to privilege-separate their applications.

The state-of-the-art solutions for offering rich expressiveness rely on library OSes (such as Haven [21] and Graphene-SGX [52]). Library OSes provide an abstraction of a virtualized process namespace to application code. Namespace virtualization requires emulating much of the OS logic within enclaves. This approach offers good compatibility with existing code; however, it comes at the peril of bloating the enclave TCB. Library OSes have reported TCB sizes in millions of lines code. Systems which have been formally verified have been smaller by orders of magnitude [37]; thus, library OSes are not within the realm of practical verification in the near future.

III. SOLUTION

A. PANOPLY *Overview*

PANOPLY provides a new design point in systems that enable rich Linux applications on SGX enclaves. PANOPLY provides the abstraction of a *micro-container* or *micron*, which is a unit of application logic that is enclave-bound.

A Linux process can import or host one or more microns. Microns *do not* get their own virtualized namespace, but instead share it with their host Linux process; that is, they invoke system calls (e.g. filesystem, network, and so on) just as non-enclave code in the same Linux process would. Microns do have their private address space, which is isolated in enclaves, and can share an arbitrary amount of public address space with the host process. By default, the code and data of the micron-based logic is allocated in private memory. There exists an explicit PANOPLY API by which micron can communicate with code outside.

TABLE II. POSIX API SUPPORT IN PANOPLY. COLUMN 3 DENOTES NUMBER OF APIS FOR SUB-CLASSES OF SERVICES. COLUMN 4 LISTS IF THE PANOPLY CAN GUARANTEE TO PRESERVE THE SEMANTICS OF THE SERVICE (`SAFE`) OR NOT (`WILD`).

POSIX Standard	Service Description	# of APIs	API Type
Core Services	**Process Creation and Control**	5	`Safe`
	Signals	6	`Wild`
	Timers	5	`Wild`
	File and Directory Operations	37	`Safe`
	Pipes	4	`Safe`
	C Library (Standard C)	66	`Wild`
	I/O Port Interface and Control	40	`Wild`
Real-time Extensions	Real-Time Signals	4	`Wild`
	Clocks and Timers	1	`Wild`
	Semaphores	2	`Safe`
	Message Passing	7	`Safe`
	Shared Memory	6	`Safe`
	Asynchronous and Synchronous I/O	29	`Wild`
	Memory Locking Interface	6	`Wild`
Threads Extensions	**Thread Creation, Control & Cleanup**	17	`Safe`
	Thread Scheduling	4	`Wild`
	Thread Synchronization	10	`Safe`
	Signal Delivery	2	`Wild`
	Signal Handling	3	`Safe`
Total	**Total**	**254**	

Micron code has access to a rich subset of the POSIX v1.3 API, listed in Table II which PANOPLY supports. The exposed API includes system calls for filesystems, network, multi-processing, multi-threading, synchronization primitives (via `pthreads`), signals and event management (via `libenv`). The service API's are classified as `SAFE` or `WILD` based on whether PANOPLY guarantees to preserve their semantics or not. Specifically, for `SAFE` API's PANOPLY guarantees POSIX semantics and application can assume that semantic correctness (barring aborts). The API design choice is intentionally chosen so as to eliminate much of shared libraries (e.g. `libc`) from the TCB of the enclave-enabled micron code. Figure 3 shows parts of a standard Linux application that would be in the TCB (shaded grey) in our architecture. PANOPLY embeds a thin *shim library* which interfaces with the PANOPLY API. The shim library is added to each enclave at compile-time, and is largely transparent to the developer.

Compiled micron code includes PANOPLY's shim library, which plays a key role in protecting against a malicious OS. First, the shim library acts as controller or manager and provides micron management functionalities such as support to create / destroy microns and providing identity management for microns, using SGX-specific features. More importantly, the shim code enforces a stronger integrity property across all microns in an application — it ensures that control and data-flows between microns conforms to the control and data-flow of the original application. We call this the *inter-micron flow integrity* property. The shim ensures inter-micron flow integrity by establishing an authenticated secure-channel protocol between microns automatically. To assert the importance of this

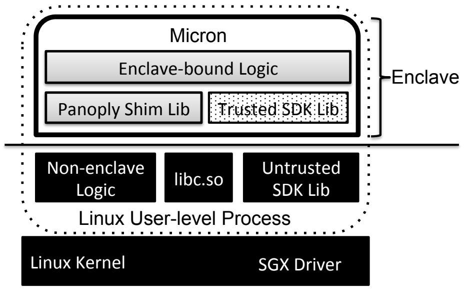

Fig. 3. Overview of codename architecture. All the regions within an enclave are trusted, the regions shaded as black are untrusted, grey shaded regions are newly added / modified as a part of codename system.

global property, we demonstrate real attacks on several case studies similar to those in Section II-B, which succeed even if memory isolation properties are enforced locally on all of the enclaves. Second, the shim library performs checks for Iago attacks [26], safeguarding against low-level data-tampering for OS services. It acts as an interface to invoke other services such as threading, synchronization and event management.

To achieve low TCB, PANOPLY chooses a *delegate-rather-than-emulate* strategy. First, PANOPLY delegates all the system calls to the untrusted OS. PANOPLY intercepts the calls to the `glibc` API, which allows us to leave the `glibc` library outside enclave TCB (Figure 3). Second, PANOPLY re-thinks the design of threading, forking and other interfaces so as to not emulate the entire OS logic in the TCB, but instead delegating it to the OS. PANOPLY delegates the scheduling logic with the underlying OS. Thus, the application scheduling is not guaranteed to be same as the original code (hence API is `WILD`). However, this trade-off is justified because OS can anyways launch a DOS attack and is in-charge of enclave scheduling. As an advantage, it allows us to place minimal number of checks within the enclave, thus significantly reducing the TCB. Lastly, PANOPLY modularly includes API calls in the enclave i.e., only the APIs which are used by a given enclave-bound code are included inside the enclave. This choice is inspired from micro-kernels to reduce the TCB.

B. Usage Model & Scope

PANOPLY consists of a set of runtime libraries (including the shim library), and a build toolchain that assists developers in preparing microns. PANOPLY takes as input the application program source code and per-function programmer annotations to specify which micron should execute that function. Thus, if the analyst wishes to partition the application into multiple microns, she can annotate different functions with corresponding micron identifiers. Functions that are not marked with any micron identifiers can be bundled and delegated to one separate micron by default. In cases that PANOPLY is not able to identify the micron for a function, it prompts the analyst for providing additional annotations or sanitization code. PANOPLY instruments the application, creating multiple micron binaries, each embedded with its own shim code. Each micron is compiled as a library package (e.g. `micron-A.so`). It consists of 3 libraries internally: PANOPLY shim library, the Intel SDK library and any other libraries that micron code uses. Figure 4 shows the schematic view of the PANOPLY

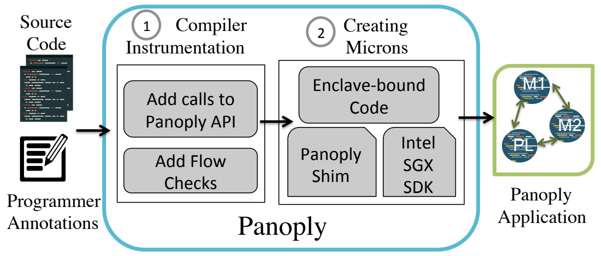

Fig. 4. System Overview. PANOPLY takes in the original program and the partitioning scheme as input. It first divides the application into enclaves and then enforces inter-micron flow integrity, to produce PANOPLY application.

system. The compilation phase adds code for inter-micron flow integrity and PANOPLY APIs.

Out-of-scope Goals. The choice of partitioning scheme is orthogonal to our work and is left to the security analyst. Existing tools for program partitioning could be leveraged here [22], [24]. Instead, PANOPLY focuses on porting the partitioned code to enclaves. PANOPLY does not reason about the functional correctness of original application implementation. Any bugs or vulnerabilities in the original application would persist in the micron-based application. PANOPLY cannot prevent denial-of-service attacks from the OS, since SGX itself does not provide this guarantee. Our system currently does not take special measures to thwart enclave side-channels. However, one can employ orthogonal defenses for enclave side-channels [30], [44]. Lastly, we blindly trust all analyst-inserted annotations and instrumentation to be secure and correct. We trust the SGX hardware, which includes a secure implementation of (a) isolated memory, (b) cryptographic attestation for enclaves, and (c) random-number generator.

IV. PANOPLY DESIGN & SECURITY

In designing PANOPLY, we aim to support essential UNIX abstractions as well as provide necessary security guarantees for single or multi-micron execution of an application. We implement several checks within each micron, which allows us to adhere to *delegate-rather-than-emulate* design decision. We describe the important design choices in PANOPLY and its security guarantees.

A. Runtime Micron Management

PANOPLY processes the source code along with user annotations to identify how the analyst wishes to separate the application logic among microns. As done in several other works [24], [35], PANOPLY partitions the micron application code accordingly. PANOPLY instruments it to output a micron binary file (a shared library) at the end. The PANOPLY shim library ensures that the final micron code supports secure micron initialization and inter-micron flow integrity.

Micron Initialization and Identity Establishment. PANOPLY creates an instance of a micron within an enclave via Intel SGX SDK API which takes the micron binary file as an input. PANOPLY generates the micron binary file based on the developer annotations provided in the source code. If the micron is created successfully, the hardware returns an identifier which is a unique value for that instance of micron. The OS assigns a

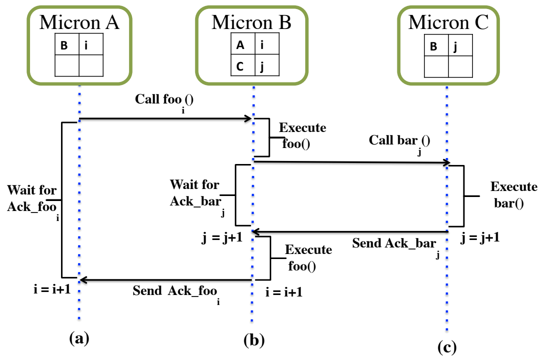

Fig. 5. (a) Micron A is a sender micron that makes a call to foo function in micron B. (b) Micron B is both a sender as well as receiver micron that executes function foo and invokes function bar in micron C. (c) Micron C is a receiver micron that executes function bar.

handle for this identifier, however PANOPLY does not trust the OS handles. Instead, PANOPLY shim library assigns a micron-identity for each micron instance and internally maintains a mapping of this name and the hardware identifier returned by SGX. PANOPLY uses this micron-identity for all further inter-micron interactions. Before starting any interactions, the PANOPLY shim code attests other microns by the processor for an attestation quote and verifies the measurement from the public attestation service [33].

Inter-micron Flow Integrity. Inter-micron flow integrity ensures that the application exhibits the same control and data flow across multi-micron execution, as intended in the original application. In our design, we consider the interaction between multiple microns as a communication protocol where the adversarial OS acts as a mediator between the microns. PANOPLY guarantees that micron execution is protected against attacks such as silent aborts or message replay from the underlying OS. For this purpose, PANOPLY enforces a secure and authenticated inter-micron protocol with the following design.

Confidentiality & Integrity: During inter-micron communication, the (mediator) OS can observe all the information that is exchanged between the microns. Moreover, it can change the values of the incoming and outgoing messages from the microns. To block this capability of the OS, PANOPLY performs authenticated encryption of every incoming and outgoing message of the micron.

Sender / Receiver Authentication: When a micron is executing, the OS can impersonate to be a sender micron and spoof spurious calls to a receiver micron (for e.g., Line 13 in Figure 1). It can also impersonate to be a receiver and hijack all the micron-bound calls. To prevent such spurious messaging from the OS, PANOPLY ensures that only a pre-defined set of authorized microns included in the application interact with each other. For this, it makes use of the secure mapping of micron identities to its instance which is established during the micron initialization phase. PANOPLY shim code checks the authenticity of each micron-identity for all `call-rets` points — only legitimate call sites can invoke the respective functions and return back. Further, it also checks if a particular micron is authorized to perform a given interaction. PANOPLY discards messages from unauthorized microns and aborts execution.

Message Freshness: The OS can replay a previously captured message to arbitrarily invoke functions of the receiver microns (for e.g., line 25 in Figure 1). To prevent against replay attacks, PANOPLY ensures that every call message from one micron to another is distinctly identifiable. This allows us to maintain the freshness guarantees for every valid message exchanged between microns. To achieve this, the sender micron generates a 128-bit fresh nonce using Intel SGX's random number generator (`RDRAND`)[1] at the start of session with each micron, as a *session-id* between a pair of microns. The nonce is incremented as a counter / sequence number in all the subsequent messages and authenticated-encrypted to prevent the OS from tampering it. Each micron stores a mapping of `micron-id` $\mapsto$ `session-id` for every micron it communicates with. Figure 5 shows the protocol for communicating between three microns. The sender attaches this nonce in the first call message. The receiver verifies if the incoming message is from an authorized sender. On validating the authenticity of the sender, it checks if it is the first message from the particular sender during the application execution. If it is the first message of a session, the receiver sets the nonce value as the session-id for communicating with the sender micron. For other subsequent messages, the receiver micron checks if the session-id is in the expected sequence. For all valid incoming call messages, the receiver micron executes the requested function and increments the session-id on successful execution of the function (Figure 5 (b), (c)). The sender micron waits for the receiver micron to complete the execution and return (Figure 5 (a), (b)). Thus, the application executes sequentially and proceeds in lock steps.

Reliable Delivery: When delivering a message from one micron to another, the OS can arbitrarily drop the call messages (for e.g., line 4 in Figure 1). This results in silent abort / failure of the communication between two microns. To avoid this, PANOPLY introduces an *acknowledgment message* similar to `ACK` in the TCP protocol. After the receiver micron executes the requested function in the call message, it appends the session-id with an ACK message to the sender micron and increments the session-id. The sender micron checks whether the session-id attached with the ACK is valid. If the check is successful, the sender micron increments its own session-id and is ready to make the next call message to the receiver micron (Figure 5 (a), (b)). If the sender fails to receive an acknowledgment, it successfully detects that the OS has silently dropped its messages. The sender micron aborts its execution if the ACK is either not received or contains an invalid session-id.

B. *Expressiveness with Low TCB*

Enclaves are limited in terms of the programming expressiveness they support. Specifically, standard C primitives such as system calls, networking APIs, file system operations, multi-threading, multi-processing and event handling do not work out-of-the-box within enclaves. As a recourse, PANOPLY addresses each of these limitations by creating microns.

System Calls. Enclave code cannot directly make a system call to the OS. Therefore, PANOPLY redirects all the system

[1]This number is passed through an extractor to generate a cryptographically secure random number.

calls from the enclave code to our custom wrappers in the PANOPLY shim library. The shim library is responsible to make the correct `OCALL` and `ECALL` to invoke these calls in the OS. It is in-charge of exiting out of the enclave, executing the system call in the untrusted component and relaying the return values back to the enclave.

Inside the shim, PANOPLY performs custom checks on the system call return values inside the enclave to defend against well-known class of Iago attacks [26]. Nearly two third (205/309) of the system calls return 0 or error, and another one third (104/309) return an integer. In addition to return values, system calls can also return data via parameters passed by reference. Most of the parameters are data structures which contain control fields. After checking the specifications of all the system calls, we identify 50 system calls (16%) that write information into 20 distinct structures. Most of them (18 structures) only have integer field types, similar to return values. Other structures contain structure pointers or function pointers. Specifically, we add sanitization code to ensure that return values are consistent with the POSIX semantics.

For return values that have static data types such as integer fields with 0 or error calls, PANOPLY compares the return value with 0 and a set of valid error numbers per system call. If it detects return values such as invalid error numbers, then the check reports failure. For return values that lie within a range, PANOPLY checks that the return value conforms to a valid range depending on the return type. For e.g., the `read` system call in Linux has an integer return type as shown below.

```
size_t read(int fd, void *buf, size_t len);
```

The shim code knows the input length (`len`) requested by the application to read. Thus, it checks that the system call return value and pass-by-reference parameter (`buf`) is less than or equal to the requested length. This check limits the return value within a much smaller range than the original range of variable type. Similar to return values, PANOPLY sanitization logic checks all the structures with integer fields returned by the system call. The policy is similar to the check on return values — based on the field type, the code ensures that the value is within a valid range.

Few of the system calls return structure pointers or function pointers which are handled specially in our system. For e.g., `connect` system call returns a pointer to a structure of `sockaddr` type as shown below. For such dynamic data types, PANOPLY library needs developer assistance for performing deep checks for such structures. For example, PANOPLY can check the fields of `sa_family_t` structure which itself is a field of `sockaddr` that is returned by the `connect` system call with developer annotations for the correct bounds.

```
int connect(int sockfd,
      const struct sockaddr *serv_addr,
      socklen_t addrlen);

struct sockaddr {
   sa_family_t sa_family;
   char      sa_data[14]; }
```

Note that PANOPLY's checks do not protect against any vulnerabilities such as buffer overflow that are present within the original application. For example, in the `read` call if the `len` is not provided, then a check for the `buf` variable is not performed by PANOPLY since the buffer size is not known statically. The developer can provide an upper bound based on the maximum file size. However, since there is no concrete check on the size of `buf`, the original code itself is susceptible to buffer overflow attacks.

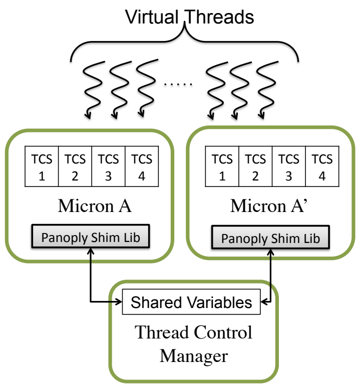

Fig. 6. Design for on demand multi-threading in PANOPLY.

Multi-threading. Enclaves have a limited support for threading — there is a gap between program abstractions in UNIX and that provided by SGX. In SGX SDK, the application has to statically determine how many and where the threads are initialized. Intel SGX ensures that all the thread-local memory (such as thread stack) is isolated from each other. It also provides a Thread Control Structure (TCS), a data structure to hold thread-specific information such as program counter, stack pointer, register values, and execution context state of the thread. Since the enclave has to know the total size to be allocated for saving TCS structures, the enclave has a statically pre-specified number (say, k) of TCS data structures. Thus, it can only support maximum k concurrent threads during the entire execution of the enclave. Thus, SGX does not allow to dynamically execute arbitrary threads on-demand. Although the application can create threads outside the enclave, the number of concurrent threads executing in the enclave at a given time is limited to the pre-determined value. Many applications do not fit in this regime of thread usage. For instance, number of concurrent threads in a web server is a function of number of requests that the server has to serve.

On-demand Threading: PANOPLY gives the abstraction of arbitrary number of threads by supporting POSIX threading (`pthread`) API. Applications can dynamically create threads by calling the standard pthread API. To support this abstraction, PANOPLY realizes the notion of virtual threads for the applications. Figure 6 shows an overview of PANOPLY design to support on-demand multi-threading. Under the hood, it multiplexes these virtual threads on the underlying enclave threads and uses the TCS structure and SGX threading APIs. Specifically, PANOPLY threading API creates TCS structures for a pre-determined pool of threads per micron (say k). When a specific thread wants to enter an enclave, PANOPLY first checks if the host-enclave can accommodate the thread concurrently. If so, PANOPLY sets the correct micron id, thread arguments, and redirects the thread execution to enter the micron. The challenge is when a micron has already reached its maximum concurrent thread limit. At this point, there are two design choices: (a) evict an executing thread and schedule the execution of the new thread (b) spawn a new micron to cater to the $k+1$ thread. Although eviction is a clean solution, it incurs additional operations of save-restore and scheduling inside the enclave. PANOPLY does not adopt this design and instead spawns a host enclave on-demand. Specifically, if the micron reaches its maximum capacity of concurrent threads, PANOPLY launches a new micron thus increasing the size of threads from k to $2k$. This design choice comes with a caveat that all the shared memory between threads has to be accessible across the two microns. To this end, PANOPLY introduces a *thread control manager* for such global thread memory that acts as a reference monitor. For supporting $(k+1)^{\text{th}}$ thread, PANOPLY launches a new micron and all the shared memory operations are performed via the thread control manager, as done in a write-through cache. The PANOPLY shim library is responsible for redirecting the micron code to this thread control manager if the total number of concurrent threads running is greater than k. When the threads are less than k, they access the local copy via shim within the host enclave.

Synchronization: SGX SDK only supports basic thread synchronization primitives — spin locks and mutexes. However, higher level synchronization primitives such as semaphores, conditional variables, barriers, and read / write locks are not yet supported in SGX. Thus, the programmer has to realize these constructs by using mutexes available with the limited SGX threading support. Instead, PANOPLY exposes the full suite of `pthread` synchronization primitives inside the enclave. It implements these operations using the SGX's mutex synchronization support. PANOPLY supports the POSIX threading API `pthread` for semaphores, conditional variables, barriers, and read / write locks operations based on SDK-supported mutexes. For all synchronization operations which are local to a micron's code, PANOPLY keeps the synchronization objects inside the micron. For global synchronizations across microns, PANOPLY creates a notion of inter-micron locks. The thread control manager micron holds and manages all such shared objects and all global changes are synchronized via this micron. Specifically, to either release or get hold of a lock, a micron has to invoke the thread control manager. The value of the global objects can only be changed by a well-defined set of microns, thus enforcing the right semantics of mutexes for the objects.

Multi-processing. PANOPLY supports applications which use the UNIX `fork` and `exec` APIs. For a fork system call, PANOPLY library instructs the OS to launch a new untrusted child process. This child process then creates a new micron with the same code as the parent micron. The PANOPLY shim library performs micron initialization step and assigns a micron-identity to this micron instance. Next, PANOPLY establishes a communication channel between the parent and all the children microns for maintaining the inter-micron flow integrity. As per the POSIX `fork` semantics, the child must replicate the data memory state of the parent micron. There are three possible strategies to ensure that the child micron has access to the parent's data.

Strategy 1. A straightforward way is to do a full replica of parent micron's data and communicate it to the child micron over a secure communication channel at the fork call. This approach involves excessive micron-micron data copy and slows down the application needlessly, even when the child does not use all the parent data.

Strategy 2. A second alternative is a copy-on-demand strategy, similar to copy-on-write optimization for fork calls in model Linux implementations. To achieve this for microns, we can implement a page-fault based on-demand data passing from parent to child micron. Specifically, the parent replicates and communicates its data to the child if and only if the child accesses it. To achieve this, PANOPLY can mark all the parent micron's pages as read-only (copy-on-write) and register custom page-fault handlers [2] inside both the microns. Whenever the child faults on a data page, the custom handler can request the parent to communicate the data. We point out that the support for registering page fault handling within enclaves is not currently enabled in the SGX hardware. To overcome this limitation, we can modify the OS to notify the micron on faults. The OS is not trusted to reliably uplift the fault to the user level. However, even if the OS suppresses faults, it does not weaken the security as the parent's pages will still be sealed.

Strategy 3. A final strategy is to statically identify what data is accessed by the forked child micron. When the application performs a fork, the parent micron can replicate / communicate these statically identified data values to the child micron. To reason about the incompleteness of the static analysis, the system can raise a run-time error to flag any data values which were not replicated. This way, the developer can add custom code to replicate these values. In our experiment applications, we observe that most of the data that is required by the child micron is small and does not need full data space replication. Thus, PANOPLY can statically identify the variables and communicate their values at fork and replicate them in the child micron. Developer can additionally annotate data variables that are shared between parent and children microns. In many cases, strategy 3 is cleaner.

PANOPLY provides a generic implementation using strategy 1, such that any application which uses fork can continue to execute without excessive developer efforts. In Section IV-D, we discuss the details for implementing strategy 1 in existing SGX SDK. In the future when the SGX2 hardware is available, strategy 2 can also be integrated in PANOPLY, subject to specific design details of the SGX hardware. Currently for cases where performance is critical, developers can chose to resort to strategy 3.

In the case of `exec`, PANOPLY requests the OS to create a new untrusted process which launches a new micron for the exec-ed code. This micron code is attested by using the SGX hardware primitive before the execution begins. The shim library assigns a micron identity to this micron for further interactions with other microns. For `exec`, static identification and copy-on-demand strategy saves a lot of redundant copy operations, since the child micron doesn't need access to parent data. For cases where strategy 3 is not sufficient, PANOPLY resorts to using strategy 2.

Shared Memory. To support shared memory between two or more microns, PANOPLY establishes a shared secret between

[2]With SGX2 extensions, the hardware can directly notify the enclave when the enclave code incurs a page fault along with the faulting virtual address, page permissions, fault type and the register context [16], [39].

these microns using secure inter-micron protocol. The microns then use a public page as their shared memory resource. Every microns seals the content and writes to the public page. Rest of the microns unseal the page content within their own address space. The sealing is standard authenticated encryption with version control for preventing replay attacks [46]. This mechanism creates a notion of shared memory, such that each micron internally maintains a private copy of the memory and all changes are synchronized and broadcasted by writing sealed data to the public page.

Event Management. PANOPLY applies the delegate strategy to support event based-programming inside the enclaves. SGX hardware natively supports synchronous and asynchronous exits from the enclave. For synchronous and asynchronous exits, the hardware saves the execution state before exiting the enclave, and restores it back when the enclave resumes. For example, if the enclave receives an interrupt, the hardware performs an Asynchronous Enclave Exit (AEX), saves the current processor state into enclave memory, enters the Interrupt Service Routine (ISR) and then finally restores the processor state for resuming the enclave execution. In this case, the OS is responsible for executing the ISR and scheduling back the enclave execution. The hardware ensures that the OS cannot tamper any context registers during the exits. Thus, it is safe to delegate the event listening and notification tasks to the OS.

PANOPLY uses the OS primitives of signals and interrupts to register event listeners, callbacks and add dispatch handlers. Internally, PANOPLY hooks the underlying OS APIs and interfaces via `OCALLs`. Thus, PANOPLY enables event wait/notify mechanism, polling, event buffering, signal handling for enclaves which are necessary for event management. Our API is expressive to support event libraries such as `libevent`, `libev`, in addition to hand-coded event loops.

C. The PANOPLY *Infrastructure*

To use PANOPLY, akin to several partitioning tools [22], [24], [35], the developer marks all functions to denote which micron executes these functions. In case of our example in Section II, the functions called on Line 5, 13, and 25 are marked to be executed in a priviledged micron by annotations. Rest of the functions in the program are explicitly marked to be executed in a non-priviledged mircon.

After establishing the set of all such micron-bound functions, PANOPLY analysis identifies the inter-micron interaction boundaries. Specifically, it constructs the program dependence graph consisting of control flow and data flow. The control flow graph comprises of micron function nodes connected by call edges. Data flow graph comprises of all the parameters passed between microns and any shared memory such as global variables. For cases where PANOPLY cannot precisely identify the control and data-flow graph, it prompts the user to specify the intended flows by adding annotations.

PANOPLY instruments all the boundaries of the micron i.e., the entry and exit points of the micron wherein the control starts or ends the micron-bound execution. Each `ECALL` to be executed inside the micron is mapped to an entry point and a return point. For all such entry points in the micron, PANOPLY adds code to check the caller's identity as well as the caller's state. At all the exit point from the micron, PANOPLY adds the correct target function to be executed in another micron, along with the its current state. Hence, the adversary cannot tamper the caller (checked at the entry) or forge the callee (created at the exit). PANOPLY replaces all calls to non-micron code (such as system library APIs) with `OCALLs`. Further, each `OCALL` interface is instrumented with a set of checks on input and output call parameters as discussed in Section IV. These checks are best-effort, and the developer can add call-specific sanitization logic at any of these interfaces.

For supporting multi-threading, PANOPLY needs to know the set of memory that is concurrently accessed by multiple threads. The developer can annotate all the such variables which are subject to operations from multiple threads. In our current implementation, we manually identified concurrent memory accesses for our case studies. In the future, PANOPLY can perform precise pointer analysis to aid the developer in identifying the complete set of corresponding operations on shared memory addresses. Once the shared variables and operations are identified, PANOPLY redirects the access to all the shared variables, via the shim. The shim code is responsible for deciding if the operations on shared variables are to be performed locally to the micron or are to be carried out as global operations in tandem with threads in other microns. We follow similar strategy to mark and mediate access to shared memory and fork-exec process memory semantics.

D. Implementation

We implement PANOPLY on top of the Intel SDK [8] 1.5 Open Source Beta shipped for Linux Kernel v3.13. PANOPLY comprises of a set of API libraries and build extensions. For our case-studies, we annotate the application code to mark which functions should execute in which micron. Then we modify the application `Makefile` to use PANOPLY extensions and library. We add the interface file (`.edl`) for specifying all the enclave entry-exit points. The Intel SDK `edger8r` tool uses these files to generate boiler-plate code stubs for enclave. PANOPLY then adds specific checks to each of these stubs.

We implement the support for multi-processing (specifically, `fork` and `exec`) by patching the SGX SDK [6] and the corresponding SGX linux driver [7]. By default, when a new process is created by fork, the SGX kernel driver data structures which map the enclave virtual addresses to the EPC physical addresses for the parent process are copied over to the child process. Hence, when the child process spawns its enclave, the driver reads the data structures and assumes that the VA space and the EPC PA addresses are already taken up. However, in reality these are stale mappings from the parent enclave. If left unmodified, the SDK driver spawns the child enclave in a new VA space, which does not overlap with the existing mapping. This leads to a mismatch in the child and parent's VA layout. To work around this, our driver code ensures that the start virtual addresses of all the children enclaves is same as that of the parent enclave. Specifically, when spawning a child enclave, PANOPLY ignores the VA-PA mapping inherited from the parents enclave. Further, we ensure that the child's enclave starts at the same virtual address as the parent's enclave. Once the address layouts are identical, PANOPLY shim code in the child enclave reads the sealed contents (BSS, data, heap and stack sections) saved by parent enclave and updates its own corresponding sections.

V. Evaluation

In this section, we show the effectiveness of PANOPLY by porting popular real-world applications to microns and testing them against a suite of application-specific benchmarks. Specifically, we aim to evaluate PANOPLY for the following:

- **Expressiveness.** Is PANOPLY successful in supporting expressive programming constructs inside microns?
- **Stronger Security.** Does PANOPLY enable stronger security guarantees for applications?
- **TCB.** How much TCB reduction does PANOPLY achieve over Library OSes?
- **Performance.** How does PANOPLY perform compared to Library OSes?

All our experiments are measured on Inspiron-13-7359 machine with 6th Generation Intel(R) Core(TM) i7-6500U processor and 8GB RAM. We configure the BIOS to use 128 MB memory for SGX EPC. We use Linux 1.5 Open Source Beta version of Intel Software Guard Extensions SDK, Intel SGX Platform Software (PSW), and driver on Ubuntu Desktop-14.04-LTS 64-bits with Linux kernel version 3.13. All our applications are compiled with GCC v4.8 and built for SGX hardware pre-release mode (`HW_PRERELEASE`) with default optimization flags and debug symbols. To measure various statistics at run-time, we use Intel VTune Amplifier with SGX Hotspots analysis which is configured with the standard parameters for SGX. All performance measurements are reported over an average of 5 runs.

A. Case Studies

We select 4 application case-studies which highlight the advantages of multi-micron architecture in applications. We successfully demonstrate that PANOPLY can enable end-to-end guarantees rooted on SGX primitives as building blocks and support expressive programming constructs. The case studies include:

- Tor v0.2.5.11, where multiple directory servers form a distributed network
- H2O v2.0.0 webserver, with privilege separated support for Neverbleed [5]
- OpenSSL v1.0.1m library-as-a-service which can be imported by any application
- FreeTDS v0.95.81 database client application

1) **Stronger Security Property — Tor:** Tor is an anonymous communication system that routes the client request through a circuit of (three or more) nodes. Tor is an open distributed network which uses a directory protocol to maintain a global view of "good" (or non-blacklisted) nodes in its network. All the routers in the network periodically send their status to a directory authority server (DA). Each DA collects these status reports to generate its local view of the network. DA servers then run a form of consensus protocol to agree upon the set of global network nodes, wherein each DA sends a signed network status vote object to peer DA servers. Various real-world attacks have targeted DAs by compromising either their private keys or forging status votes to create a dishonest view of the network, thereby causing malicious or blacklisted Tor nodes to be accepted in the network [10].

Goal: Our goal is to secure the Tor DA protocol against a malicious OS on the DA servers. Specifically, we want to ensure that even if 8 out of 9 DA servers are compromised, a blacklisted node is not accepted in the network's "good" view. The attacker can only shut down the network. This high level property is hard to achieve without the use of byzantine fault tolerance protocol [25], [27], which are bandwidth hogging and thus are not used in this application. Previous work by Jain et al. has only looked at a low-level property of protecting the DA server's private keys using SGX [32]. They port only the key related operations to enclaves, and execute the rest of the code outside the enclave. While this ensures that the attacker does not get direct access to the key, it does not guarantee that Tor will always maintain the true state of the network.

The Tor DA servers act as nodes and the messages they exchange act as control and data flow edges of the blacklisting consensus protocol. Even though each DA server's secrets are secured by enclaves, it does not ensure the integrity of interactions between the DA servers. Specifically, in our extended technical report [45], we demonstrate 4 concrete attacks, similar to our running example in Section II. The attacks use the following strategies to disrupt the consensus protocol between DA servers:

- Call tampering to change the network status votes
- Force silent failures, leading to vote withholding
- Replace messages to allow compromised certificates
- Replay compromised public keys

On the other hand, if we view the protocol amongst these distributed nodes as a single process executing in a single contiguous isolated memory, then the correctness of the consensus follows directly. In this case, all the messages (such as votes, and status objects) are generated and passed without any tampering from the byzantine adversary. There are only two things that the adversary can do — abort the process or proceed. This property holds true because the adversary cannot tamper the execution flow inside the isolated memory.

If we move the entire logic of each DA server to a separate micron (i.e., 9 microns for 9 DA servers), then all the messages generated inside the micron are ensured to be untampered, they are executed in an isolated memory. Further, by the virtue of inter-micron flow integrity, PANOPLY ensures that the control and data flow of inter-micron interactions (inter-DA interaction in this case) is preserved. Thus, PANOPLY allows us to achieve the same security guarantee as executing the entire Tor blacklisting consensus protocol in a single contiguous isolated memory. As discussed above, this ensures that the byzantine adversary can only abort the consensus, it cannot bias it. We implement this architecture for Tor with the help of PANOPLY, as discussed in Section V-B.

2) **Supporting Privilege-separation — H2O:** In 2014, the Heartbleed bug in OpenSSL's implementation of TLS protocol lead to leaking server's private key due to a buffer over-read. Webserver implementations such as H2O HTTP2 server have taken precautionary measures by separating the RSA private key operations in a privilege separated process since version 1.5.0-beta4 [5]. This minimizes the risk of private key leakage in case of vulnerability such as Heartbleed. In PANOPLY design, the RSA key operations are moved to a separate micron (RSA micron) whereas the rest of the webserver code executes in another micron (H2O micron). The RSA micron comprises of functions to load the private key and to use it for encrypting

/ decrypting a given buffer and to sign the contents of a buffer. The H2O micron invokes the RSA micron functions to perform functions such as `rsa_sign`, `rsa_priv_enc` and `rsa_priv_dec`. PANOPLY ensures that the RSA keys are safe in one micron different from the rest of the application. The application micron invokes the key-storage micron via PANOPLY's interface.

3) **Supporting Enclaved-libraries — FreeTDS & OpenSSL:** FreeTDS is an open source implementation of the TDS (Tabular Data Stream) protocol which allows programs to natively talk to Microsoft SQL Server and Sybase databases. It links to OpenSSL library to support SSL/TLS for its traffic. The running example in Section II demonstrates how PANOPLY enables to split the FreeTDS logic inside two microns.

OpenSSL is a widely used open-source library for SSL/TLS protocols. It provides client and server-side implementations for SSLv3, TLS along with the X.509 support needed by SSL/TLS (`libssl`). Further, the core library also implements basic cryptographic functions (`libcrypto`). Several large-scale real life applications require SSL/TLS support for networking with clients, servers or peers. Hence, we use PANOPLY to execute OpenSSL inside a single micron. This allows large scale applications to use OpenSSL library by executing it inside the micron. It is up to the application's security architect to decide if the application code executes in the same micron as OpenSSL or as in a separate micron. For our case study we take a sample certificate verification application shipped with Intel SGX SDK [8]. The application comprises of two parts: (a) IO operations to read a list of certificates from an input file and display the results (b) X.509 certificate verification by using the root-certificate. We separate the application into two microns — one for IO and one for OpenSSL library. The OpenSSL micron uses programming constructs such as threading and network APIs. The IO micron uses file system and standard IO APIs. Further, the attacker can attack the micron-micron interactions by replaying or dropping the messages as outlined in our running example in Section II. These attacks trick the application to accept invalid certificates, thus violating the higher level guarantee of accepting only valid certificates. PANOPLY enforces inter-micron flow integrity and ensures that all such attacks are thwarted.

B. Porting Effort

We make an average change of 905 lines of code per application to port it to PANOPLY [54]. We test our applications with their regression test suite to ensure that our porting does not break the application.

Tor. Tor code uses 4 libraries — OpenSSL's `libcrypto` and `libssl` for SSL/TLS protocol, `libevent` for event handling, and `libz` for compression. We port all 4 libraries along with Tor code to microns. In doing so, we make use of PANOPLY's threading, multi-process and networking APIs. We create two microns as per the design discussed in Section V-A. We make a total of 2685 LOC changes to the complete codebase with the help of PANOPLY. To test for correctness, we ensure that micron-code passes all the 32 tests in the regression suite shipped with Tor code. We evaluate Tor with a private network comprising of 3 DA servers and 3 routers using Chutney [1]. All the nodes execute on a single machine and use local attestation. We chose the same configuration as evaluated by previous works [32], [36]. We observe a maximum of 2 threads executing in parallel for our configuration.

H2O. We configure H2O web server with Neverbleed plugin, YAML parser and the in-built event-loop implemented by H2O. We further statically link H2O with OpenSSL library which executes inside the webserver micron and includes the networking and multi-threading module of PANOPLY. We make 154 LOC changes to the code to achieve this. The application exposes 3 functions for inter-micron invocation. Further, we evaluate H2O performance with `h2load` which is a benchmarking tool for HTTP/2 [4]. We observe a total of 2 inter-micron messages and 128 PANOPLY API calls at run-time for H2O.

FreeTDS. Our FreeTDS application uses OpenSSL library, along with threading and network support from PANOPLY. We make 473 LOC changes to the application to compile it with PANOPLY. We configure FreeTDS client application with a remote Microsoft SQL database server. Our benchmark makes 48 queries (1 create, 46 insert and 1 select) to the remote database server and collects the response time. For this workload, FreeTDS makes 3 inter-micron calls.

OpenSSL. We port the OpenSSL library including the cryptographic utilities (`libcrypto`) and the TLS/SSL implementation (`libssl`). We re-use the partially ported OpenSSL code available with the Intel SDK which only ports `libcrypto`. Specifically, we enable the SSL protocol and the support for executing OpenSSL engines inside the micron. We use the PANOPLY networking and file system APIs to achieve this and change 307 LOC. We test the OpenSSL library with its regression suite, and use the `speed` and `tspeed` utility to benchmark the performance of our OpenSSL library. We observe a total of 8 PANOPLY API calls at run-time along with 1 inter-micron call.

Comparison to Graphene-SGX. We attempted to port our 4 case-studies to Graphene-SGX, by following the public instructions [3] available at the time we performed our experiments. We report that we were not able to port 3 out of 4 applications to Graphene-SGX— the applications either crash or suspend during execution. Since there is no public companion report, but only a large-scale system, it is hard to evaluate the design reasons of why the applications crash. After further investigation of the Graphene-SGX source-code, we were able to narrow down 2 cases which are problematic: (1) the `fork` semantics break in H2O which launches neverbleed in a separate process (2) the `epoll` call does not receive the socket events, thus the applications such as Tor and FreeTDS never progress. Thus, our evaluation comparisons are best-effort. We have informed the authors to implement possible mitigations in their system for these issues.

C. Reduction in TCB

PANOPLY achieves 2 orders of magnitude lower TCB as compared to state-of-art approaches such as Graphene-SGX [52] and Haven [21]. The size of compiled microns in MB is an orders of magnitude smaller than Graphene-SGX.

PANOPLY **TCB.** PANOPLY adds an average of 19.62 KLOC per application which is 5.8% of the original application code.

TABLE III. TCB EVALUATION. THE TABLE SUMMARISES THE TCB COMPONENTS OF PANOPLY AND GRAPHENE-SGX IN TERMS OF LOC AND SIZE OF COMPILED ENCLAVE BINARIES. COLUMNS 2-5 REPORT THE BREAK-DOWN FOR THE LOC OF EACH COMPONENT INCLUDED IN THE MICRON, AND COLUMN 6 HIGHLIGHTS THE PERCENTAGE INCREASE IN THE TCB DUE TO PANOPLY. COLUMNS 7-13 REPORT THE LIBRARIES INCLUDED (✓) IN THE COMPILED ENCLAVE BINARY, AND A ✗DENOTES THAT THE MICRONS DO NOT REQUIRE THE LIBRARY. COLUMN 14 DENOTES THE TOTAL SIZE OF COMPILED MICRONS. COLUMNS 15-18 DENOTE THE COMPILED SIZE OF VARIOUS COMPONENTS INCLUDED BY GRAPHENE-SGX TCB. COLUMN 19 REPORTS THE TOTAL SIZE OF COMPILED ENCLAVES FOR GRAPHENE-SGX.

Case Study	PANOPLY **TCB in LOC**					**Split-up for** PANOPLY **TCB (in MB)**								**Graphene (in MB)**				
	Enclave Logic	**Boilerplate Code**	PANOPLY **Logic**	**Total**	**% Inc**	**libssl (0.765)**	**libcrypto (3.7)**	**libz (0.137)**	**libyrmcds (0.06)**	**libyaml (0.185)**	**libevent (0.579)**	**SDK (0.05)**	**Enclave Size**	**PAL**	**Enclave**	**Trusted Libs**	**SDK**	**Enclave Size**
OpenSSL	256987	9004	10425	276416	**7.56**	✓	✓	✗	✗	✗	✗	✓	**5.88**	1.84	0.067	64.58	0.05	**64.69**
H2O	414918	9189	10425	434532	**4.72**	✓	✓	✗	✓	✓	✗	✓	**7.98**	1.84	16	64.70	0.05	**80.75**
FreeTDS	297108	9788	10425	317321	**6.80**	✓	✓	✗	✗	✗	✗	✓	**6.08**	1.84	1.2	64.58	0.05	**65.83**
Tor	436385	8817	10425	455627	**4.41**	✓	✓	✓	✗	✗	✓	✓	**13.18**	1.84	7.0	63.94	0.05	**70.99**

TABLE IV. PERFORMANCE EVALUATION. COLUMNS 2-5 REPORT THE STATISTICS ABOUT PANOPLY MICRONS. COLUMNS 6-9 DENOTE THE CPU CYCLES CONSUMED BY PANOPLY. COLUMNS 10-12 DENOTE THE EXECUTION TIME FOR EACH CASE STUDY AND THE CONTRIBUTION OF PANOPLY TO THE TOTAL TIME.

Case Study	**No. of Microns**	**Inter-Micron APIs**	**No. of `OCALLs`**	**No. of `ECALLs`**	**Split-up of CPU Cycles (in billions)**				**CPU Time (in seconds)**		
					Non-Micron / SDK	**Create Delete**	**Enclave Logic**	**Total**	**Empty Enclave**	PANOPLY	**% Increase**
OpenSSL	2	1	23695	1	0.25	6.06	0.11	6.43	2.79	3.16	**13**
H2O	2	1	124287	5	0.35	17.08	6.11	23.54	6.56	8.79	**34**
FreeTDS	2	1	86198	1	0.47	22.31	0.07	22.64	8.60	8.74	**1**
Tor	6	30	286459	5	6.65	11.77	2.58	17.41	4.54	6.72	**48**
										Average	**24**

TABLE V. BREAKDOWN OF LOC AND TCB SIZES OF GRAPHENE-SGX COMPONENTS. GLIBC LOC AND SIZE COMPUTATION ACCOUNTS FOR THE WHOLE CODEBASE COMPRISING OF ALL LIBRARIES AND PLATFORM-SPECIFIC CODE.

Component	**LOC**	**Size (in MB)**
glibc	1156740	56.9
libPAL-LinuxSGX	16901	0.9
libPAL-enclave	33103	0.2
SGX SDK	119234	0.5
Total	**1325978**	**58.5**

Columns 2-5 in Table III shows the LOC split-up for each case-study. Out of the total LOC included by PANOPLY in each micron, 9.2 KLOC (46.3%) of code is automatically generated boilerplate code to facilitate `OCALLs` and `ECALLs` mechanism. This code is generated by the Intel `edger8r` tool, which takes in a `.edl` interface file and creates stubs for passing parameters across micron boundaries. We believe that this code is easy to automatically verify in the future. In terms of compiled code size, each micron binary includes the original application logic, corresponding application libraries, the trusted libraries added by Intel SDK and PANOPLY shim library. Columns 7-14 in Table III show the split-up for the micron binary size. Note that PANOPLY selectively adds a library to a micron, if-and-only-if the micron code needs it. More importantly, PANOPLY does not include system libraries such as `libc`, `libdl`, `libpthread` in the micron.

Comparison to Graphene-SGX TCB. We refer to the public release of Graphene-SGX [3] to compute the total LOC of system. Table V shows the LOC of each component in Graphene-SGX, and the corresponding binary size. Graphene-SGX library comprises of a Platform Adaptation Layer, Linux library, and system libraries required to support a pico-process. In terms of LOC, this amounts to an increase of 1.326 MLOC. When compared with PANOPLY, this amounts to 2 orders of magnitude (127.19×) larger TCB. Note that the TCB and binary sizes are dependent on the application. We port our four case studies to Graphene-SGX and measure the total size of the binaries, since counting per-binary LoC is more complicated. Columns 15-19 in Table III lists the breakdown of size of each component. On an average, Graphene-SGX pico-process binaries are an order of magnitude larger as compared to PANOPLY microns. Haven reports a TCB of millions of lines of code in the paper [21], however we do not have access to the system to directly compare with PANOPLY.

D. PANOPLY *Performance*

PANOPLY adds a 24% CPU overhead to the application's execution time (Table IV) on an average, over the baseline cost of creating empty enclaves. It achieves comparable performance as Graphene-SGX, with PANOPLY's overheads higher by 5-10%. PANOPLY strictly prioritizes TCB over performance, and does not include any optimizations such as buffering. However, future systems can improve over PANOPLY by incrementally adding optimization with careful considerations for TCB bloat.

Performance Breakdown. We measure the micron execution bottlenecks as well as the breakdown for the total number of CPU cycles it consumes. (Table IV) Column 6-9 present the average number CPU cycles that each application consumes for various operations during its entire execution. Our preliminary performance measurements identified that bulk of the CPU cycles are spent in the Intel SDK. On further investigation, we identified three main factors which contribute to this.

First, the operation of creating and destroying enclaves takes up 6-7 billion CPU cycles for various sizes of enclaves. We launch each of the application enclaves and destroy them without executing any logic inside the enclave. Column 7 in Table IV denotes the number of cycles to do this for each specific case-study. These operations are performed by the Intel SDK, and are not an artifact of PANOPLY's design. Thus, 96% of the CPU cycles are due to enclave launch and tear-down (Column 7 vs Column 9 in Table IV). Taking this cost as a baseline, PANOPLY has a 24% overhead.

Second, copying enclave data to-and-from non-enclave memory contributes to significant fraction of CPU cycles consumed by the application. Specifically, these operations involve encryption-decryption of data entering/leaving the enclave memory, which consumes CPU cycles. For example, in the H2O application, the number and volume of such a copy operation is directly proportional to the size of the served web-page. To test this aspect, we measure the throughput of H2O webserver by serving two sizes of web-pages: 200 Bytes, 1 KB, and 6 KB for a total of $100,000$ requests from 100 clients. As shown in Figure 7, the webserver throughput decreases as we increase the size of the web-pages. The shaded-region

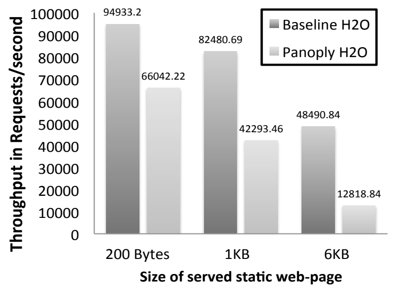

Fig. 7. PANOPLY Performance for 100000 requests of various page sizes.

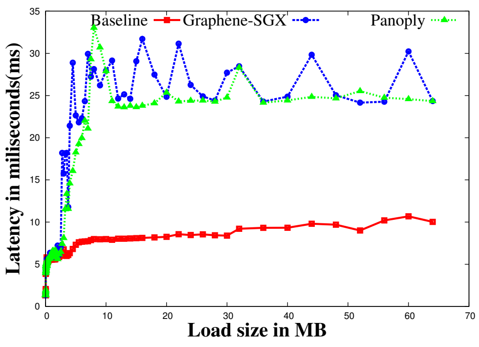

Fig. 8. LMBench Latency Performance of PANOPLY & Graphene-SGX.

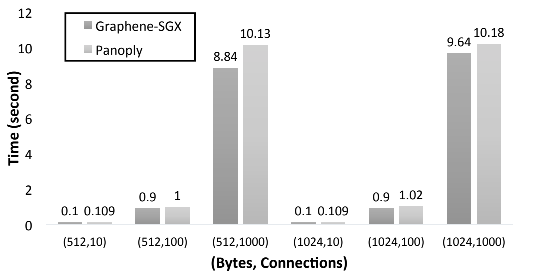

Fig. 9. OpenSSL Throughput Performance of PANOPLY & Graphene-SGX.

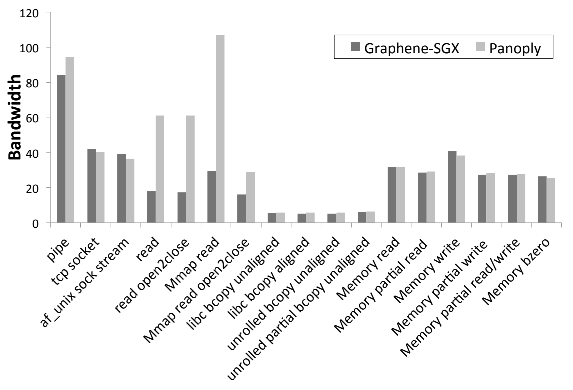

Fig. 10. LMBench Bandwidth Performance of PANOPLY & Graphene-SGX.

of the bar-graph denotes the fraction of execution time spent in the SDK routine for `OCALLs`. Thus, larger the number of `OCALLs`, the more is the overhead. As we can also see from Table IV, applications which have less number of `OCALLs`, exhibit lower overheads.

Third, all of our case studies use OpenSSL routines for various operations ranging from SSL connections to cryptographic operations. The OpenSSL library first checks if the underlying hardware supports AES-NI via `cpuid`. If it detects that the hardware supports, it uses the hardware AES instructions for its cryptographic operations, otherwise it falls back to a software implementation. In our experiments, we observe that the OpenSSL library executing inside the enclave fails to detect that our hardware has AES-NI support. Thus, it uses a slower AES routine which adds latency by consuming more CPU cycles. We suspect that the SDK has not rolled out support for executing `cpuid` instruction inside the enclave. This change was also proposed by previous work [21].

We point out that the Linux SDK itself does not use hardware AES-NI instructions for encryption-decryption. Instead it uses software implementation of AES routine, as was pointed out by recent public disclosures [34]. In our case studies, this further amplifies the slow-down of each encrypt / decrypt operation inside microns by 20% of the total execution.

Comparison to Graphene-SGX. Our design does not significantly degrade the performance as compared to Graphene-SGX. We compare the performance of two systems for executing OpenSSL. Specifically, we test varied sizes and frequencies of data written out of the micron-enclave, since it is the main factor for PANOPLY overheads. To this end, we configure a SSL client-server setup and measure the CPU execution time for 6 configurations of number of requests and size of payloads: 512 / 2014 bytes of data for each of the 10 / 100 / 1000 client connections. Figure 9 depicts the performance of Graphene-SGX and PANOPLY. For these configurations, PANOPLY has 5-10% higher overheads than Graphene-SGX.

Since comparison of a single user-application may not be a conclusive evidence for the overall performance of PANOPLY, we present a fair comparison of these two systems. Specifically, we port the LMBench benchmark to PANOPLY, which is supported [3] by Graphene-SGX as well. Figures 8 and 10 present the latency and bandwidth performance of PANOPLY and Graphene-SGX. Our first observation is that the performance overheads of both the systems are significant over the absolute baseline of a native application executing without any enclave infrastructure. This re-iterates our earlier findings that the slow-down due to enclave life-cycle operations (create and destroy) are common to any system that uses SGX. Secondly, the memory latencies exhibited by PANOPLY and Graphene-SGX are comparable. As shown in Figure 8, the PANOPLY latency is almost-always lower than Graphene-SGX, whereas both the systems exhibit similar overheads over an absolute baseline. Finally, we measure the bandwidth for various types of operations including network, memory, file IO for the two systems. PANOPLY performs comparable to Graphene-SGX for memory and network operations (Figure 10). For file-backed mmap operations, the overhead for PANOPLY is observably

[3]The Graphene-SGX system is not stable when executing the full LMBench benchmark suite, and crashes non-deterministically with segmentation faults. The results presented here are assimilated over 24 attempted runs to gather a full set of evaluation.

larger than Graphene-SGX. Since PANOPLY performs these operations via libc interface, the number of enclave entry and exits per operation is larger. On the other hand, Graphene-SGX uses a narrower interface and hence for file IO, it incurs a lower number of enclave transitions. This is one of the factors which causes the bandwidth variation for this subset of IO operations.

VI. RELATED WORK

PANOPLY is a new design point in SGX enclave design space that achieves low TCB while maintaining expressiveness for enclave-bound code. PANOPLY's inter-micron flow integrity guarantees a higher level security property, unlike previous systems which target low-level confidentiality primitives [47], [48]. We discuss how PANOPLY differs from existing systems in terms of TCB, design goals, scope and end-to-end guarantees.

TCB. PANOPLY design is driven by the delegate-rather-than-emulate philosophy, which is the key for lowering the TCB. Specifically, we do not perform namespace management inside the enclave, which is common approach for library OS designs. The goal of library OSes is to achieve a narrow ABI [21], [41], [52], so as to maintain compatibility and portability. Hence, these systems implement bulk of the system logic inside the library OS to map the system call APIs to their narrow ABI interface. In PANOPLY, we are not limited by these design choices. We expose a larger POSIX API to microns, and delegate all the system logic to underlying operating system. This is a reasonable choice because the OS can perpetrate the same set of attacks even with a narrow interface. Thus, our design choice allows us to keep system libraries such as `libc` outside of TCB, while achieving the same level of security.

Security of Single Enclave. PANOPLY is the first system to demonstrate control and data-flow attacks on enclave-enclave interactions. It prevents such attacks by ensuring inter-micron flow integrity. Recent works have pointed out that enclaves are susceptible to side channel attacks via page faults [55] and cache [28]. Currently, PANOPLY does not guarantee defenses against such side-channels. However, applications can employ off-the-shelf defenses proposed recently [29], [38], [43], [44]. Weichbrodt et al. [53] recently showed that if the enclave logic has use-after-free or TOCTOU bugs, then the OS can exacerbate the effect of these bugs to perpetrate control-flow attacks inside the enclave code. PANOPLY assumes that the enclave is free of any logic or memory bugs. Strackx et al. highlight that the adversary can shut down enclaves and abuse the execution by doing a hardware state-replay attack [49], [50]. PANOPLY can use their proposed solution to ensure hardware state contiguity in the future.

Partitioning Applications for SGX. PANOPLY enables expressive enclave-bound code with a low TCB. Thus, it can execute maximum application logic inside one or more microns while ensuring that the PANOPLY application maintains the security guarantee. However, PANOPLY leaves the choice of partition design to the security architect [22], [24], [35]. Jain et al. [32] propose the use of enclaves to protect Tor DA server keys inside enclaves to protect against well-known attacks [14], [15]. Kim et al. [36] propose designs to use SGX for networking applications such as SDN-based inter-domain routing, Tor directory servers and ORs. Atamli-Reineh et al. [20] propose four partitioning schemes ranging from coarse-grained partitioning (single enclave for whole application) to ultra-fine partitioning (one enclave per application secret) for executing OpenSSL library in enclaves.

SGX Containers & Sandboxes using Enclaves. Scone [19] is a concurrent system which uses Intel SGX enclaves to isolate docker containers running in a public cloud setting. We summarise the key design differences between Scone and PANOPLY. Firstly, the interface exposed by PANOPLY is at POSIX level, whereas Scone exposes a system call interface to the enclaves. As an artifact of this design choice, PANOPLY does not execute any libc library inside the enclave. On the other hand, Scone executes the libc library (specifically musl libc) inside the enclave. Secondly, PANOPLY does a synchronous exit for executing code outside the enclave, whereas Scone does an asynchronous exit. These two variations lead to a different design in terms of TCB, performance and system challenges. Thirdly, the on-demand threading model proposed by PANOPLY spawns new microns in separate enclaves to scale the number of threads. This way, each thread in the application is associated with a unique thread in the enclave. Scone uses a M:N threading model. Hence, when the application scales it threads, it is forced to multiplex on a limited number of existing threads in a single enclave. Lastly, PANOPLY is designed for multi-process applications, which comprise of multiple micron containers and user processes. Hence its design comprises of in-built support for fork, exec, clone and a secure communication interface between multiple microns and processes. Scone only supports applications with a single-container process running inside a single-enclave, which is a subset of PANOPLY.

Ryoan [31] is a concurrent work for executing distributed SGX native client (NaCl) sandboxes. PANOPLY's execution model of multi-micron applications varies from Ryoan, since in PANOPLY, all microns that belong to the same application trust each other. Ryoan introduces a request-oriented data model where each enclave is in-charge of processing the input only once. Ryoan ensures that each service sandbox confines the user-data to itself, while allowing mutually distrustful parties to compute over sensitive data. In Ryoan, the NaCl executes the system calls and all the buffer and file IO operations are backed by in-enclave memory.

VII. CONCLUSION

PANOPLY bridges the gap between expressiveness of the SGX-native abstractions and requirements of feature-rich Linux applications. PANOPLY offers a new design point, prioritizing TCB over performance, without sacrificing compatibility. It achieves 2 orders of magnitude lower TCB than previous systems.

ACKNOWLEDGMENTS

We thank Mona Vij and Simon Johnson from Intel for their feedback. Thanks to Chia-Che Tsai and Donald Porter for releasing code and discussions on Graphene-SGX. This research was partially supported by a grant from the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

References

[1] "Chutney - The Chutney tool for testing and automating Tor network setup," https://gitweb.torproject.org/chutney.git.

[2] "FreeTDS: Making the leap to SQL Server," http://www.freetds.org/.

[3] "Graphene-SGX Library OS - A Library OS for Linux Multi-process Applications, with Intel SGX support," https://github.com/oscarlab/graphene, (Accessed on 12/06/2016, Commit 9958214).

[4] "h2load - HTTP/2 Benchmarking Tool," https://nghttp2.org/documentation/h2load-howto.html.

[5] "H2O Neverbleed: Privilege Separation Engine for OpenSSL / LibreSSL," https://github.com/h2o/neverbleed.

[6] "Intel SGX for Linux," https://github.com/01org/linux-sgx.

[7] "Intel SGX Linux Driver," https://github.com/01org/linux-sgx-driver.

[8] "Intel Software Guard Extensions SDK - Documentation — Intel Software," https://software.intel.com/en-us/sgx-sdk/documentation.

[9] "[MS-TDS]: Tabular Data Stream Protocol," https://msdn.microsoft.com/en-us/library/dd304523.aspx.

[10] "Tor Network Is Under Attack through Directory Authority Servers Seizures," http://thehackernews.com/2014/12/tor-network-hacked.html.

[11] "Tor Project: Anonymity Online," https://www.torproject.org/.

[12] "CVE-2009-3555 TLS Session Renegotiation Vulnerability," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3555, 2009.

[13] "RFC 5746 - Transport Layer Security (TLS) Renegotiation Indication Extension," https://tools.ietf.org/html/rfc5746, February 2010.

[14] "Tor Project Infrastructure Updates in Response to Security Breach," http://archives.seul.org/or/talk/Jan-2010/msg00161.html, 01 2010.

[15] "Possible Upcoming Attempts to Disable the Tor Network — The Tor Blog," https://blog.torproject.org/blog/possible-upcoming-attempts-disable-tor-network, 12 2014.

[16] "Software Guard Extensions Programming Reference Rev. 2." software.intel.com/sites/default/files/329298-002.pdf, Oct 2014.

[17] "Intel Software Guard Extensions Enclave Writer's Guide," https://software.intel.com/sites/default/files/managed/ae/48/Software-Guard-Extensions-Enclave-Writers-Guide.pdf, 2015.

[18] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," in *HASP 2013*.

[19] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *OSDI 2016*.

[20] A. Atamli-Reineh and A. Martin, ch. Securing Application with Software Partitioning: A Case Study Using SGX.

[21] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *OSDI 2014*.

[22] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting Applications into Reduced-privilege Compartments," in *NSDI 2008*.

[23] E. Brickell, J. Camenisch, and L. Chen, "Direct Anonymous Attestation," in *CCS 2004*.

[24] D. Brumley and D. X. Song, "Privtrans: Automatically Partitioning Programs for Privilege Separation," in *USENIX Security 2004*.

[25] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *OSDI 1999*.

[26] S. Checkoway and H. Shacham, "Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface," in *ASPLOS 2013*.

[27] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults," in *NSDI 2009*.

[28] V. Costan and S. Devadas, "Intel SGX Explained," Cryptology ePrint Archive, Report 2016/086, 2016, http://eprint.iacr.org/2016/086.

[29] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation," in *USENIX Security 2016*.

[30] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang, "M2R: Enabling Stronger Privacy in MapReduce Computation," in *USENIX Security '15*.

[31] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data," in *OSDI 2016*.

[32] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. Kang, and D. Han, "OpenSGX: An Open Platform for SGX Research," in *NDSS 2016*.

[33] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel Software Guard Extensions: EPID Provisioning and Attestation Services," ser. Intel Corporation.

[34] L. M. JP Aumasson, "SGX Secure Enclaves in Practice: Security and Crypto Review — Kudelski Security," Black Hat USA, 2016.

[35] D. Kilpatrick, "Privman: A Library for Partitioning Applications," in *USENIX ATC 2003*.

[36] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han, "A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications," in *HotNets 2015*.

[37] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal Verification of an OS Kernel," in *SOSP 2009*.

[38] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," 2016. [Online]. Available: http://arxiv.org/abs/1611.06952

[39] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel Software Guard Extensions Support for Dynamic Memory Management Inside an Enclave," in *HASP 2016*.

[40] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *HASP 2013*.

[41] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the Library OS from the Top Down," in *ASPLOS 2011*.

[42] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy Data Analytics in the Cloud," in *IEEE Symposium on Security and Privacy 2015*.

[43] M.-W. Shih, S. Lee, T. Kim, and M. Peinado., "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *NDSS 2017*.

[44] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *AsiaCCS 2016*.

[45] Shweta Shinde and Dat Le Tien and Shruti Tople and Prateek Saxena, "Panoply: Low-TCB Linux Applications With SGX Enclaves," National University of Singapore, Tech. Rep., Dec 2016.

[46] Shweta Shinde and Shruti Tople and Deepak Kathayat and Prateek Saxena, "PodArch: Protecting Legacy Applications with a Purely Hardware TCB," National University of Singapore, Tech. Rep., Feb 2015.

[47] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani, "A Design and Verification Methodology for Secure Isolated Regions," in *PLDI 2016*.

[48] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani, "Moat: Verifying Confidentiality of Enclave Programs," in *CCS 2015*.

[49] R. Strackx, B. Jacobs, and F. Piessens, "ICE: A Passive, High-speed, State-continuity Scheme," in *ACSAC 2014*.

[50] R. Strackx and F. Piessens, "Ariadne: A Minimal Approach to State Continuity," in *USENIX Security 2016*.

[51] F. Tramer, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi, "Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge," Cryptology ePrint Archive, Report 2016/635, 2016.

[52] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and Security Isolation of Library OSes for Multi-Process Applications," in *EuroSys 2014*.

[53] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves," in *ESORICS 2016*.

[54] D. Wheeler, "SLOCcount," http://www.dwheeler.com/sloccount/.

[55] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *IEEE Symposium on Security and Privacy 2015*.

[56] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town Crier: An Authenticated Data Feed for Smart Contracts," Cryptology ePrint Archive, Report 2016/168, 2016, http://eprint.iacr.org/2016/168.