2020

Extracting Information From Subroutines using Static Analysis Semantics

Luke A. Burnett
Wright State University

EXTRACTING INFORMATION FROM SUBROUTINES USING STATIC ANALYSIS SEMANTICS

A Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Cyber Security

by

LUKE A. BURNETT
B.S. C. S., Wright State University, 2017

2020
Wright State University

Wright State University

GRADUATE SCHOOL

May 4, 2020

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Luke A. Burnett ENTITLED Extracting Information From Subroutines using Static Analysis Semantics BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Cyber Security.

Junjie Zhang, Ph.D.
Thesis Director

Mateen Rizki, Ph.D.
Chair, Department of Computer Science and Engineering

Committee on
Final Examination

Junjie Zhang, Ph.D.

Keke Chen, Ph.D.

Meilin Liu, Ph.D.

Barry Milligan, Ph.D.
Interim Dean of the Graduate School

ABSTRACT

Burnett, Luke A. M.S. C.S. , Department of Computer Science and Engineering, Wright State University, 2020. *Extracting Information From Subroutines using Static Analysis Semantics.*

Understanding how a system component can interact with other services can take an immeasurable amount of time. Reverse engineering embedded and large systems can rely on understanding how components interact with one another. This process is time consuming and can sometimes be generalized through certain behavior.We will be explaining two such complicated systems and highlighting similarities between them. We will show that through static analysis you can capture compiler behavior and apply it to the understanding of a function, reducing the total time required to understand a component of whichever system you are learning.

Contents

List of Figures

List of Tables

Listings

Acknowledgment

I would like to thank my advisor, Dr. Junjie Zhang, for guiding me through the process of defending my thesis and ideas. I would also like to thank Dr. Meilin Liu and Dr. Keke Chen for spending their time to review the work I have completed, critiquing this manuscript, and serving on my thesis committee. I'd also like to thank Dr. Adam Bryant for his mentorship. The help of academic staff at Wright State University is immeasurable with regard to its development of my professional life. I would also like to thank my family for their support, with countless encouragement. Finally, I would like to thank Teresa, for the love and support she has given me during some of the most difficult times of accomplishing this feat.

Abbreviations

Intel Software Guard Extensions — SGX

Trusted Platform Modules — TPM

System Management Mode — SMM

Management Engine — ME

Exception Level — EL

Application Programming Interface — API

Processor Reserved Memory — PRM

Enclave Page Cache — EPC

Memory Encryption Engine — MEE

Thread Control Structure — TCS

Architectural Enclaves — AE

Provisioning Enclave — PE

Provisioning Certificate Enclave — PcE

Quoting Enclave — QE

Translation Lookaside Buffer — TLB

Static Single Assignment — SSA

Value Set Analysis — VSA

Introduction

Trusted computing has been a focus of processor designers in order to mitigate post exploitation vectors for manipulating data that should be considered sensitive[3][4]. Computing services will continue to move away from being hosted inside businesses as the usage of cloud computing technology grows in popularity. One study predicts that as much as 94 percent of workloads will be processed in the cloud, with 73 percent being accessible by public cloud instances[5]. over 3 billion mobile subscriptions exist globally, with each continent continuing to grow in mobile network size[6]. The computing devices that touch our lives have individual approaches to processing sensitive data.

When examining these interfaces from a security perspective, determining how private information is transfered between parts of a system is integral for deciding how mindful the engineers were when designing these processors. We will examine two separate, but popular, architectures and their respective trust interfaces: Intel Software Guard Extensions(SGX), and Arm TrustZone. The purpose of these sections are to give examples of systems that are complex in nature.

Afterwards, we will discuss generalizing semantics of subroutines to extract information. While not in the context of the previously discussed sections, the semantic discussion would ideally aid someone examining these technologies. The background of both TrustZone and SGX as surfaces for examination aim to show the complexity of just these two systems.

1.1 Motivation

Applications and internet software is increasingly growing with the rise of internet of things devices entering the market. With every new device on the internet there are more opportunities for invasions of privacy[7], malicious actors commandeering your devices[8], or manufacturers making your device prematurely obsolete[9][10]. Understanding how these devices work is important for any number of reasons, but this process can be time consuming.

Additionally, desktop applications are becoming more increasingly connected with the addition of Trusted Platform Modules(TPM)[11]. This TPM hosts an external processor that can be used to encrypt and decrypt data. This discrete processor can be used to protect your computer, or be used by a malicious actor to send a encrypted malware that is theoretically impossible to examine[12].

Understanding a new code base can be difficult, even when source code is present. This thesis aims to provide a guided introduction to automated static analysis techniques and how to apply them to your understanding of a foreign interface. This thesis will also show how automated semantic understanding of a binary can assist in your understanding of a binary.

1.2 Goals

During the development of this thesis we have developed a system to work with NSA's Software Reverse Engineering suite, Ghidra[13]. This system is able to integrate semantics that are understood by reverse engineers as they work through understanding executables from any processor supported by Ghidra. This framework operates on P-Code[14], the intermediate representation that gets lifted from the disassembly process of a subroutine within a binary.

The framework is designed to be easily extendable and have an API for resolving the estimated values of variables at a given point in time using a limited value-set analysis technique[15]. The framework provides the ability to access basic blocks and automatically be called back with a p-code operation to be handled, allowing for concurrent analysis to hypothetically be run in conjunction with one another.

This current implementation aims to resolve type sizes and potential subtypes based on how structure members are used within a set of functions. This problem seemed to be the easiest to show such a framework could be worthwhile, in the author's opinion. Time taken to resolve ”easily shown” dependencies can be tedious and repetitive, so reducing the amount of time spent on this task can be important. This framework also aims to show that semantic extraction based on intermediate representation is as valid as native instructions for extracting semantic facts from lifted disassembly.

1.3 Related Work

Some authors have tried to integrate semantic understanding to their static workflow. In 2012 Rolf Rolles gave a keynote[16] in which he compiled several different works to describe semantic understanding within a binary, and how semantic understanding of a binary triumphs over simply syntatic understanding of a binary when certain goals are desired. This proposal explores deobfuscation of x86 binaries with the understanding of a specific type of anti-debugging technique. This approach utilizes IDA[1] and the Hex-Rays decompiler addon in order to retrieve the intermediate representation of a binary instead of parsing the x86 syntax. Unfortunately this type of representation, as far as the author of this thesis knows, is available for the supported Hex-Rays platforms. This number of platforms is much smaller than the number of supported platforms for Ghidra.

Edward J. Schwartz et al[17]. discuss a semantic based approach(called OOAnalyzer) for discovering objects and their inheritance relationships within a binary. Using a framework for deriving facts about a binary, observed semantics of a binary are used to uncover

[1]https://www.hex-rays.com/products/ida/

type relationships in portable executable binaries, a common executable format for Microsoft Windows. OOAnalyzer searches for common patterns used by C++ compiled binaries, like the explicit return of the first parameter passed to a function, usage of the `this` object as if it were an object, virtual function tables. This approach is limited to the specific types of binaries that can be analyzed by the ROSE compiler infrastructure[17].

Blazytko, Contag, and Aschermann[18] wrote about using semantics to generally deobfuscate and derive the behavior of code that has been purposely obscured. This technique relies on using the input and output behavior of code they want to synthesize. With this, synthesis is then applied to a search algorithm to determine behavior of the system when it is not purposely obfuscated. This technique relies on instrumenting the binary in order to obtain an execution trace. This technique is great for understanding obfuscation as a black box but can fail if the technique used to hide data relies on side effects or is computationally expensive.

In 2014, Andrew Ruef introduced software[19] that specifically uses IDA to lift semantics from defined instruction sets (ARM, x86/x86_64) to VEX, an intermediate representation used by popular software Valgrind[20]. This tool has a dependence on precondition of knowing the calling convention of whatever architecture you are targeting for analysis. This technique also relies on VEX, which can normally be obtained through dynamic analysis using tools like Valgrind or through concolic analysis using angr[21].

1.4 Organization

The first chapter of this thesis describes the problem, desired outcomes as a result of this work, approaches using dynamic analysis, and why static analysis can be preferred over dynamic analysis. The second chapter has a background section on interfaces that are complex which would be ideal for static analysis but would be poor targets for dynamic analysis. The third chapter describes vocabulary and defines information used in our work. The fourth chapter describes our approach to solving this problem. The fifth chapter evaluates our approach on effectiveness. The sixth chapter states conclusions and postulates how this approach could be applied to other problems in this space.

Historical Information

2.1 Intel History and Overview

Intel processors, upon starting, cause the processor to be placed in *real mode* if one of the later described modes isn't entered before the first long(or far) jump. A long/far jump is an instruction using both a segment register and an offset with a `jmp` instruction[22]. Real mode addressing involves using a *segment register* along with an offset for accessing resources[23]. The formula for determining the accessed memory when using the segment register and offset is:

$$memory = (register_{segment} * 16) + offset$$

Although this equation suggests that we have only 2^{20} bytes available to us, we have slightly more due to the way that setting certain registers work. We can access even more memory addresses by utilizing manufacturer features and setting the A20 address line for more feature rich booting systems, eventually leading to accessing the rest of the address space and eventual entry into *protected mode*. Real mode is responsible for bootstrapping hardware (excluding enclaves), like high speed memory devices (random access memory), graphics controllers, networking interfaces, and other system modes that handle management interrupts.[1] After configuring system hardware for usage the processor will be placed into *protected mode* if the processor supports 32 bit mode or *long mode* if it supports 64 bit mode.

[1]Intel specifically mentions in [23] that this type of initialization is somewhat dependant on the processor. This makes sense and is, therefore, difficult to generalize.

Protected mode and long mode allow for us to utilize functionality that we are accustomed to in more accommodating tasks like user space application development. These modes allow for us to access the respective address spaces like virtual memory. Interestingly, the Intel processor starts execution at the highest mebibyte of memory[23].[2] This is to accommodate systems that have smaller amounts of RAM without needing to understand how much RAM is installed in the system. Having the boot firmware located in the highest mebibyte of memory allows for more efficient management of physical memory, due to the chunks of memory not being split preemptively due to system constraints placed by the manufacturers.

Virtual memory allows systems to effectively overlay an address space for separate programs in an operating system. Each process has a set of entries that keep track of which physical pages that process has access to. This allows for the pages of memory to be written back to disk in the instances where a process isn't scheduled to run as often, or saving memory when a large set of processes happen to use some of the same underlying pages. For example, processes share the same copy of the C shared library. Page tables aren't visible to the process and is typically presented with a flat memory model. Virtual memory became a hardware feature of Intel processors in 1985[24].

An additional feature of paged memory allow for specific regions of the address space to have permissions associated with the purpose of the page. Having granular permissions for a region of memory allow for operating systems to hinder the utilization of weird machines[3] within your system[25].

As systems became more complicated, it was increasingly important to define boundaries for system responsibility. Some components of a computing system are more privileged than others, meaning they aren't as restricted in modifying the system. More privileged segments traditionally have an easier time with hiding and abstracting details away

[2]See the *Power-Up (Reset Vector)Handling* section for this particular tidbit.

[3]https://en.wikipedia.org/wiki/Weird_machine

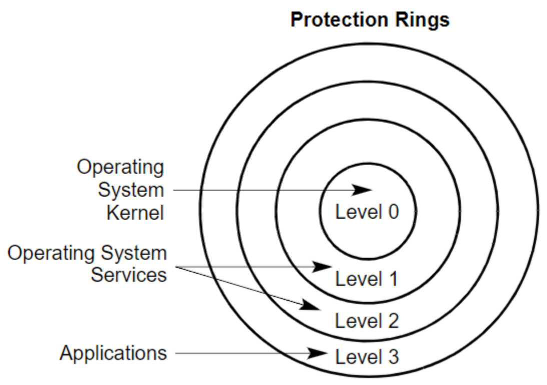


Figure 2.1: Intel Privilege Rings[1].

from less privileged segments of the Intel system. Intel processors refer to privilege segments as *rings*2.1. Intel's ring system are based on the idea of *hierarchical protection domains*[26]. People are most familiar with are ring 3 and ring 0, the application and kernel space respectively. In general, rings that are numerically lower than another ring in comparison can leverage its privilege to perform stealthy actions that would be considered malicious if the behavior is controlled in the right manner.

This still isn't the entire picture. If you have ever had a processor overheat or some other obscure problem with your home computer, you may have encountered an event that was not handled by the operating system. Just as interrupt events are handled by different portions of your operating system (or user application), events can generated by your processor that are handled inside of *System Management Mode*(SMM)[27]. This mode of operation cannot be masked by the operating system and handling of the interrupt is transparent to the other portions of the system that the interrupt even occurred[28]. SMM can manipulate hardware freely and examine contents of the processor modes above it. SMM is commonly referred to as Ring -2.

SMM Isn't enabled at boot-time like one would expect. SMM has its own portion of RAM (SMRAM), as well as a reserved address space for accomplishing its goals. SMM is enabled after the system BIOS has finished initializing just enough of the system to service management interrupts. A management interrupt can stay pending until it is acknowledged by the management interface[28].

Below the System Management Mode is the Intel *Management Engine* (ME). The ME runs on a separate microprocessor[29] that is largely responsible for performing management tasks. The firmware running on this microprocessor is proprietary and obfuscated to hide it from outside analysis. The ME is running while the processor has access to power, meaning that you can access certain functionalities of the computer even while the main is in an ”off” state. Since the ME isn’t a state of processor execution but is necessary for a complete view of the processor with regard to privileged attack surfaces in Intel hardware, ME is addressed briefly.

Each privilege level is designed with a specific purpose in mind. By abstracting each level above the lowest level, you can implement for granular hardware capabilities while supporting the higher layers with appropriate abstractions that avoid regression. Each layer has a job that it needs to complete and in some instances it can be difficult to achieve transparency effectively when working with state outside of a processor. For example, understanding that hardware not located inside of the processor itself can have state that is modified by a layer/ring before the appropriate/dispatched service gets to view that hardware’s interrupt. If a service needs to examine information, it is free to view and manipulate the data of the protection rings above it. This presents problems with hosting your proprietary or private data in cloud computing environments. Having your data and infrastructure hosted at a remote location can be quite beneficial for companies that cannot afford (or have the desire to) build infrastructure to host their own computing resources.

As computing resources became cheaper to produce, the need to host infrastructure outside of businesses also came to a head. The need to scale up with business while reducing staffing costs, along with the cost of high speed internet getting cheaper, necessitated the rise of large servers emulating hosts for business needs. These pieces of functionality are supported by both software and hardware. Hardware allows for the operating systems that businesses interface with to be unmodified, effectively creating *ring -1*[30]. The software that runs in ring -1 manages resources between the different operating systems, also known as guests, letting one computer transparently run several operating systems.

Since individuals are running their proprietary applications inside another person's computer there is a demonstrated need to hide some things from the underlying layers of your system. Imagine if the cloud hosting provider you are utilizing to process sensitive customer data was able to silently skim data for its own nefarious deeds. One way to ensure that our data is unable to be seen from more privileged modes of execution on the processor is to utilize hardware functionality specifically designed to restrict data from being seen from outside the correct context.

Intel's SGX is designed to reduce the attack surface within an application. Privacy inside a binaries address space is achieved throuh dedicated hardware that is bundled with processors that support SGX[31]. Instead of having the weakness of being at a lower execution level than the kernel, hypervisor, or management software, SGX allows for code inside of an untrusted environment to at least have some level of accountability, as long as you implicitly trust Intel to implement the primitives. The area in which private details are stored is called an *enclave*. The technology that performs trusted actions on behalf of the manufacturer is called *trusted execution technology*[32].

Intel's SGX sits alongside ring 3, at the same level as the application layer[33]. This is a peculiar choice, but effectively follows the separation of privileges between layers and avoids getting more privilege than it would need. SGX is not initialized at system runtime, and is instead created when a service running on the processor requests an enclave. The initial state of the enclave is loaded from storage[33]. Afterwards, a program may place whatever it likes inside the enclave, code or data, and then tell the enclave to perform actions on the program's behalf.

The astute reader may be wondering how someone could trust the execution state of the enclave if it is loaded on demand from storage. The program that is fetched from memory may be corrupted or tainted from a malicious actor. Intel allows whomever is communicating with the enclave to have the secure processor provide a checksum of sorts to prove that the enclave is in a state that makes sense. This is referred to as an *attestation key*[34]. The attestation key is provided by Intel and is the designated standard for determining if your enclave is in a predefined, trusted state.

2.2 ARM History and Overview

ARM Processors start in a trusted mode, which is referred to as the *secure world*[35]. In order to fully explain the ARM boot model, privilege explanation will need to happen first. For the purpose of this thesis, I will address ARM processors that inherently support the trusted mode of execution. (TrustZone) ARM works on the concept of *exception levels*. There are four defined exception levels, each level having its own purpose within the context of the system itself.

Before discussing the ARM boot process, understanding the privilege structure is important. ARM processors have two security states, secure and non-secure. Each exception level can operate in a secure or insecure context in the newest version of the ARM specification, while in older ARMv8.3 chips there is no secure exception level 2[36]. The ARM processor can only switch from a non-secure to secure (or vise versa) state from exception level 3, whose responsibilities will be explained later[37]. Exception levels can only be advanced one at a time and must be passed directly above or below the current state of execution. ARM processors start in a privileged mode, similar to Intel processors.

The secure monitor is the gateway between the secure and insecure worlds in ARM processors[37]. This interface is similar to the system procedure call in operating systems, where an unprivileged process wants to communicate with a privileged process in order to have something done on the unprivileged process's behalf. This is all done at *exception level 3*(EL3), or if AArch32 EL1. This is the highest privilege level and allows the processor to place itself into a secure mode. The monitor code is largely vendor implemented and can vary depending on which manufacturer wrote the firmware for your processor.

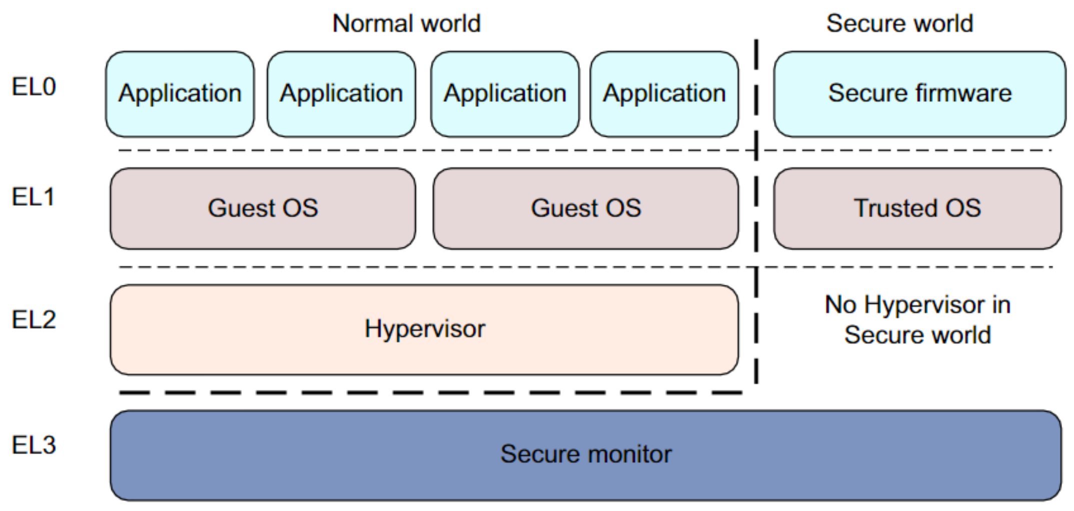


Figure 2.2: ARM Exception Levels[2]

After the secure monitor is installed at boot, it will set up other exception level states. The only mandatory exception levels are exception levels 0 and 1, the application and kernel space respectively. In a system where these are the only exception levels initialized, the highest exception level is responsible for hardware management[38]. The secure monitor will initialize the state for exception levels 1 (and 2 if it will be utilized) in the non-secure world. After allowing the secure kernel at exception level 1 to initialize its own state, the secure monitor will give execution to the highest non-secure world component. This may be a hypervisor (exception level 2) or a kernel (exception level 1).

As stated before, ARM processors are segregated by exception level. In ARM parlance, exception level 0 is similar to ring 3. Figure ARM Exception Levels[2] shows the exception levels available to ARMv8 processors. When you take into consideration that there is a separate mode for secure applications, the secure exception level 0 is used for trusted applications that can be specified for specific use and purpose for the system. This could include any level of separation within the system, like a filesystem wrapper or the ability to have some cryptographic function executed in the secure world.

The secure operating mode of the processor allows for trust to be segregated by applet.

Each trusted action has its own application space similar to a binary running on your box. Applications like fingerprint reading, retrieving credit card data from secure memory, and NFC payments[39] can all be separate processes inside of your trusted application layer. The only difficulty here is that each application has to communicate through a well defined communication protocol defined by ARM. Using the exception raising programming interface defined by ARM, you can build your own functionality.

For example, say that a process running at exception level 0 wants to access some important data that is protected by TrustZone. That process will need to raise an exception that can either be handled at that same exception level or raised. Raising the exception through each utilized running exception level allows the call to reach the secure monitor, and the monitor will decide if it wants to transfer your data to the secure world and propagate your request to the proper trusted applet.

One peculiar fact to be observed about processes running in the secure processor state is that each process is able to read (or write) any portion of the processor[40]. This is an odd security boundary to allow breached, as a hijacked trusted application can completely rewrite how each portion of the secure world. There was a mitigation developed and implemented in specifications for ARM processors. Until ARMv8.4, there was no secure exception level 2. By introducing a secure exception level 2, each application located in the secure space can receive its own segregated program space, further reducing the attack surface introduced by having secure layers be implicitly trusted. This allows for individual processors to implement sandbox-like features for each applet, mitigating the ability to modify contents within other secure applet spaces.

Example Systems

As discussed, both Intel and ARM processors have interfaces for storing and processing information inside of an area that is purportedly secure. These interfaces rely on vendor implementations and can be large enough to include mistakes. We can't naturally assume that programs which we trust can be implicitly trusted. Security researchers should spend the time to understand these interfaces. This section will identify how Intel SGX and ARM TrustZone are communicated with on at an assembly instruction level. This section will also try to further explain the platform differences.

Taking time to examine these interfaces can be time consuming, as often the interface for acquiring information from parts of the system can be either proprietary or poorly documented. Reverse engineering sections of a binary for one platform may not be useful for generalizing information across several platforms. This can be arduous and produces diminished results. This section serves to describe the purpose of common interfaces and utilizing these similarities to discover how vendor implemented APIs can be discovered using automated program analysis techniques.

Luckily, for researchers looking to quickly identify which services on their machines may or may not be utilizing these processor features, by disassembling executables on the system we can narrow down the surface we need to examine. By looking for these markers, we can examine what data may be flowing in and out of the secure processor zones. First, we must discuss in greater detail the process for utilizing each secure enhancement before discussing how one would automatically characterize APIs.

3.1 Applications and SGX

One of SGX's primary goals includes establishing trusted computing resources within a remote, untrusted, environment. SGX operates at the same ring of privilege as applications launched by unprivileged users. Remote trust is established in a few ways. When first entering and utilizing a piece of trusted hardware, you must trust the manufacturer of that hardware.

When initializing an enclave, the processor will create a memory region called the *Processor Reserved Memory* (PRM)[41]. PRM is initialized and reserved early within the boot process of the processor and is yielded to the processor. Any attempt to modify or read this memory range causes a protection fault. Inside the PRMs data region is the *Enclave Page Cache* (EPC)[41], which holds a mapping of physical memory pages mapped to enclave-only access. The PRM is small in size and is large enough to hold a structure similar to traditional page tables for secure processes so you can dynamically release and retrieve resources for enclave purposes. Each controlled page of memory is encrypted and kept track of using the *Memory Encryption Engine*(MEE)[42]. The MEE subverts normal memory access operations by translating memory accesses to the pages corresponding to secure enclaves when requested from the correct context.

Each secure enclave operates similar to a process within the context of an operating system, with a few exceptions with regard to book keeping. An enclave has its own set of saved registers for securely saving and restoring state before entering and exiting the enclave area. Enclaves support multithreading, which means that enclave threads have their own version of thread local storage, called the *thread control structure*(TCS)[43]. Enclaves have metadata associated with the individual enclave instance, like a cryptographic hash representing the contained data and the total size of said data area. Enclaves must also implement their own internal memory management operations (like a heap manager).

An application can load whatever program it likes into memory one page at a time. Adding a page of memory will change the metadata of the enclave, referred to as its *mea-*

sure[44] in Intel nomenclature. After all pages necessary for an enclave to function are added to the secure address space, the enclave is finalized. Before entering the enclave, the processor will save the state of the current process and then turn off precise event based sampling[45], which would give someone unnecessary insight into the enclave. This allows the enclave to appear as a single instruction to performance monitoring tools. If an interrupt is issued to the process while the enclave is executing, the enclave will exit and save its context related to the TCS[45]. This context will be populated with routines that properly handle saving the state of the enclave as well as the structure used to resume execution of the enclave. Enclaves may be saved to disk by performing an action called *sealing*[46]. Sealing ensures that data integrity of the enclave is kept. when communicating with a remote enclave, the process for determining if the state of the enclave is pristine is referred to as *attestation*.

For all intents and purposes, attestation is also referred to as *remote attestation*[47], as some key components rely on communicating with Intel in order to ensure the integrity of the enclave. SGX's attestation process is based off of a process called *direct anonymous attestation*[48], which tries to ensure that the identity of the owner of the platform is preserved while still being able to prove the authenticity of the underlying hardware. This protocol is a group scheme and allows for trusted third parties to authenticate platforms without any knowledge of who the platform may be owned by. The protocol for attestation is not necessarily important for understanding the interfaces utilized by system implementors and won't be covered in excruciating detail.

After the remote party communicates with the application endpoint, the application will speak with the quoting enclave. The quoting enclave will take some information from the enclave that is being attested, as well as a public key from the person requesting the attestation of the enclave being verified. This information is wrapped up inside of a structure called the *REPORT* structure, allowing for verification to occur[49]. When the quoting enclave receives the REPORT, the quoting enclave will sign said REPORT with its Enhanced

Privacy ID(EPID) key, turning the REPORT into a *QUOTE*. The QUOTE is then sent back to the requesting application and then finally sent off to the *Intel Attestation Service*, which is the only service that can validate EPID keys from quoting enclaves[34].

When an application is using an enclave to protect its memory, there are several ways to interact with the trusted portion of the processor. An enclave can share memory with the calling process in order to perform actions with protected data, referred to as an *enclave call*, or ECall[50]. ECalls implicitly enter the enclave and allow it to execute. As the enclave is executing it may need to call untrusted functions outside the enclave, which are referred to as *outside calls*, or OCalls[50].

Enclaves are built with the idea of abstracting responsibility for enclave services to the same trusted code base that would be used to bootstrap trust on the processor. Intel provides a sample base set of services that should be used to provision enclaves and manage your trusted code within the system. Intel provides a driver that correctly communicated with the trusted hardware, some drivers/shared object files that help your program utilize well defined interfaces, operating system services that can be used to communicate with enclaves, and some example ”service” enclaves that perform additional duties outlined in the Intel manual and briefly here. Collectively, these are referred to as the *Architectural Enclaves*(AE)[51].

One of Intel’s provided enclaves, the *Launch Enclave*(LE), helps generate launch tokens for enclaves. This token will allow different enclaves to execute and can help enforce policies for the type of enclaves that are launched or generating security info for the enclave as it runs[52]. The LE has a list of policies kept by Intel for other Architectural Enclave permissions.

The *Provisioning Enclave* (PE) participates in provisioning the EPID key of an enclave container, as well as managing attestation of a machine. It will encrypt attestation keys for other enclaves before they are sent out to a separate entity requiring that an enclave prove that they are trustworthy[52]. This enclave works closely with the *Provisioning Certificate*

Enclave (PcE) in order to get the processor certificate[53]. This certificate can only be retrieved from this enclave.

Intel's *Quoting Enclave* (QE) responsibly retrieves the signed attestation key and then verifies that the key is legitimate. The QE will then produce a structure that can be remotely verified by a party wishing to ensure that the integrity of the system is present[54]. The report produced by this enclave can be verified with Intel's servers and determine if the client certificate stored inside the processor is revoked or not, in order to provide a mechanism for replacing compromised or insecure certificates[55].

3.2 Areas of Interest for Intel SGX

From a reverse engineering standpoint, understanding what data is manipulated by a proprietary enclave implemented by a vendor is interesting. Understanding which instructions utilized by the processor handle data that we are interested in can help us identify the starting point for identifying information to provide context to how a program operates. This section will outline processor instructions and their purpose in the context of an Intel enclave.

When an application is setting up an enclave, they will need to add pages to the enclave page cache in order to protect their data. This is accomplished by issuing an *EADD* instruction. This instruction will add a page to the EPC associated with a particular enclave[56].

An application may also want to evict a page from the EPC as it may not be needed or it may also want to obfuscate what is located inside the enclave by confusing someone who is disassembling their application. This instruction's mnemonic is *EREMOVE*[57].

More importantly, a page may be ”blocked”. This would cause a page fault whenever that particular page is accessed in the future, even by the owning enclave. Pages that are blocked are marked for eviction from the enclave. In order for a page to be removed from the EPC it must be prepared for removal (by blocking it) and then have no translation lookaside buffers(TLB) pointing to the page. This instruction's mnemonic is *EBLOCK*[58].

There is an accompanying instruction to load a page and simultaneously be blocked, whose mnemonic is *ELDB*. There are instructions for explicitly loading a page as unblocked as well, with the mnemonic *ELDU*[58].

By comprehending the usage of an instruction we should be able to construct some useful state that is kept track of by the client utilizing the enclave functionality in the same manner that ARM's TrustZone instructions will present a similar state outline. This state will be useful for recovering type information about the API we may be trying to discover.

3.3 Applications and TrustZone

Arm's TrustZone technology allow for system integrators to choose exactly what they may need in order to protect their intellectual property. Arm provides an implementation specification for system designers and the companies designing the system-on-a-chip can fabricate the processor in a manner that they see fits. With this in mind, system designers can add their own modifications to the processor, with regard to hardware or software, as long as the processor follows the specification outlined by Arm. This means that adding pertinent details with regard to implementation can be difficult as they may vary between manufacturers. This section will outline more specifically how the processor utilizes advertised trust functionality, highlighting an open source implementation of the secure world processor. This particular open source TrustZone implementation utilizes the Linux kernel. There is an ARM for Windows implementation, but this will be outside the scope of this thesis.

ARM processor's trusted computing implementation has the ability to communicate and transfer information to the secure portion of the processor (the secure world). Since the secure world essentially mirrors the insecure world, propagating information into the secure portion of the processor involves having that data handled at each privilege(exception) level. In the Intel enclave's, the secure portion of the processor is isolated from the rest of the processor and ensures that data inside of the enclave cannot manipulated/viewed by portions of the processor that may be privileged. ARM TrustZone does not enforce this strict separation[40], but as a result will need to transmit the data between the different insecure layers in order to reach the secure monitor.

When an application wants to utilize the trusted portion of a processor, this is referred to as ”taking an exception”. As the processor handles a privileged exception[59], the only options that exception level has are to handle the exception or raise the value[60]. When raising exception levels, the ARM processor has a specific register that saves the return address for the level that raised the exception in a dedicated register, the *exception link register*(Just like how the regular link register is used to save the return address for a subroutine)[61]. An exception can be *taken from* any exception level, but you can never *take an exception to* EL0[62].

According to Arm’s ARM specification, insecure EL0 does not take exceptions[60]. The job of taking an exception is left to EL1 (and if implemented, EL2) and properly delegated to EL3, known as the secure monitor. This equates to communicating some piece of information to a special device file from your user application. Within the Linux POSIX layer is a special type of file that can be accessed and used in order to achieve our goal of transmitting information from a user owned process to the kernel[63]. Implementing the layer from EL0 to EL1 as a kernel driver which creates a device file allows for the kernel to be allowed to be implemented as a proprietary interface without directly interfering with how the kernel normally handles files or devices that are populated within the filesystem.

3.4 Areas of Interest for TrustZone

When attempting to call a function in a higher exception level, an instruction will be called with the proper target exception level. The **S**uper**v**isor **C**all (SVC) instruction targets EL1[64]. The **H**ypervisor **C**all targets EL2[65]. The **S**ecure **M**onitor **C**all targets EL3[66]. These instructions cause the processor to go to the processor’s exception vector and then enter the software interrupt state. This state has access to the same registers that the caller uses to make these instruction calls. Implementation of the ARM system determines how the arguments for this interface are extracted, by typically they are extracted from registers as the interrupts have access to the same registers that the caller can manipulate before issuing the interrupt. This adds an extra challenge to determining the API usage, as we need to understand the calling convention of the target API.

Apart from the call itself, the system implements the API target and how each of the exception raising instructions works with the system. Understanding how these system calls are implemented is paramount. Just observing the arguments to the instruction itself, a single immediate value is used for each instruction. This makes it very attractive for system designers to implement handling each of the privileged instructions as a C switch statement, after extracting extra arguments from the registers passed by the caller. This is ideal for determining which functions correspond to the arguments that are passed to the trusted applets.

Statis Analysis Semantics

Machine language and the underlying processor are tied together by semantic understandings of how to best translate high level programming constructs into assembly instructions that are most efficient for that platform. Understanding how a subroutine translates to a higher level function's constructs can be time consuming. Methodology exists to attempt automatically translating machine language to a language that most individuals are comfortable with reading. This process is called *decompilation*. This will not be the focus of this section, but many of the techniques discussed are suitable for decompilers.

All the techniques discussed are forms of *static anaylsis*. This refers to the fact that no portion of the code that is analyzed actually is ran, but only examined and determined to have certain qualities. First we will discuss some terminology and then we will discuss methodology for determining how the trusted interfaces can be examined to determine what you are able to pass to the trusted portions of the processors. This section will also introduce terminology that can be used to discuss how compilers construct subroutines and the terminology for discussing different techniques that are used throughout the compilation process, and what that may mean for the function we are looking at.

4.1 Disassembly Terminology

When examining a binary there are a few ways for someone to determine the functionality and purpose. This section will focus exclusively on executable blobs and vocabulary used to discuss the purpose of a given subroutine within a binary. There will not be a focus on executable binary formats, aside from discussing how they may aide us in uncovering information about the binary's purpose and how it may be used as part of the larger system.

4.1.1 Dominators

The *dominator* of a graph is important to understand from an analytical standpoint. A node A inside a graph is dominated by node B if every directed edge of the graph from the entry point to A goes through node B[67]. Understanding how a node inside of a graph dominates another node is important for comprehending something called *single static assignment*[68] and how it can help us estimate the value of a register at any given point in time. Another term to be familiar with is the *immediate dominator*[69], which is the "closest" dominator to note A. That is to say, node B is an immediate dominator of node A if no dominators exist between A and B.

4.1.2 Function Prologue

Typically, when a a subroutine is entered, there is some agreement between the caller and the callee that some set of registers that are used are saved and some will be modified. This agreement is discussed in the form of a *calling convention*. Some architectures will have a fairly common calling convention, while other binaries that they interface with use a separate calling convention. These differences are important for determining functionality.

Due to the nature of preserving registers, the callee will sometimes choose to place the registers it needs to save on the stack before performing any functionality that is encapsulated by the subroutine. This can be helpful for determining what type of code is being dealt with. For example, when looking at code that was written in C++ typically the "this" object will be saved at some point before calling some subroutine, if that subroutine happens to be a member of the object's class.

4.1.3 Basic Blocks

One of the most common units when looking at a disassembled function in an analytical manner is the *basic block*. A basic block is a series of instructions that have a single entry point and a single exit point. The entry and exit may have several target destinations but the important property provided by the basic block is that the instructions are executed together. Basic blocks may include subroutine calls.

Combining basic blocks together with control flow instructions can create a graph representation for a subroutine that can be used to describe and understand the behavior of a function without the source code. It isn't necessary to convert functions into this form if you are personally reading the disassembly, but having the disassembly separated into basic blocks can be exceedingly useful from a static analysis standpoint.

4.2 Static Single Assignment Form

Static single assignment form(SSA) provides a more mathematical representation to a disassembled subroutine. When working with most high level languages (like C, C++, etc) you can declare the existence of a variable and assign several values while it is in the correct scope. With SSA, a variable is given a single value for the entirety of its lifetime and cannot be given any other values[68]. When referring to a specific instance, we can guarantee that this value is always the same given the circumstances it was derived. This single assignment style is similar to proclaiming that the value is fact. When the value of a register may contain several possible values, this is referred to as a *phi* value.

When looking at a variable in a programming context, assignment is a proposition to say that an identifier is equal to some concrete value at assignment (or evaluation-time if the language happens to be lazy). The value inside the context of SSA is always assigned to a single instance of a given variable. The variable will only ever be one the value that is assigned to it. Combining this with the concept of the basic block, we can infer what a register's value may be given the context of some basic block. If we can infer the value of a register within a basic block we can use that information to more accurately understand subroutines that we don't yet have a type signature for.

4.3 Value Set Analysis

Inside programming languages like C/C++, the use of memory for generic programming constructs is almost mandatory if you would like to do anything useful. The fact that memory is malleable within a system provides usefulness for engineered products but can also cause grief if those systems are not accessible to inquiring minds. When translating high level structures to architecture specific assembly you may find the use of a value to offset your structure to get the member you are interested in. This is because the data you are accessing conforms to the structures type, telling the compiler where the data you are interested in is at.

With each subroutine, we have a set of values that we can analyze, hence the name of this methodology. Using *value set analysis*(VSA) we can partially correlate functions that may have not been related before by combining values grazed from subroutines and how those values are used in context. This can give us an idea as to which subroutines work with which objects in a binary that was written in C++.

4.4 Program Slicing

When determining what may have influenced a piece of data at the time of a subroutine's invocation it will be easier to infer the type of the argument if the irrelevant context is removed. When approximating the value in argument registers you can calculate exactly what a register is going to be by looking at the operations performed on the register. Removing operations that are not necessary for inferring the value contained within a register is referred to as a *program slice*. A program slice gives context to a value.

Normally, program slicing applies to source code. A slice is defined by a slicing criterion, a combination of a statement and the variable you are slicing. Examining statements that may be executed before the criterion statement and evaluating if that statement determines if the statement is in the program slice.

4.5 Challenges with binary decidability

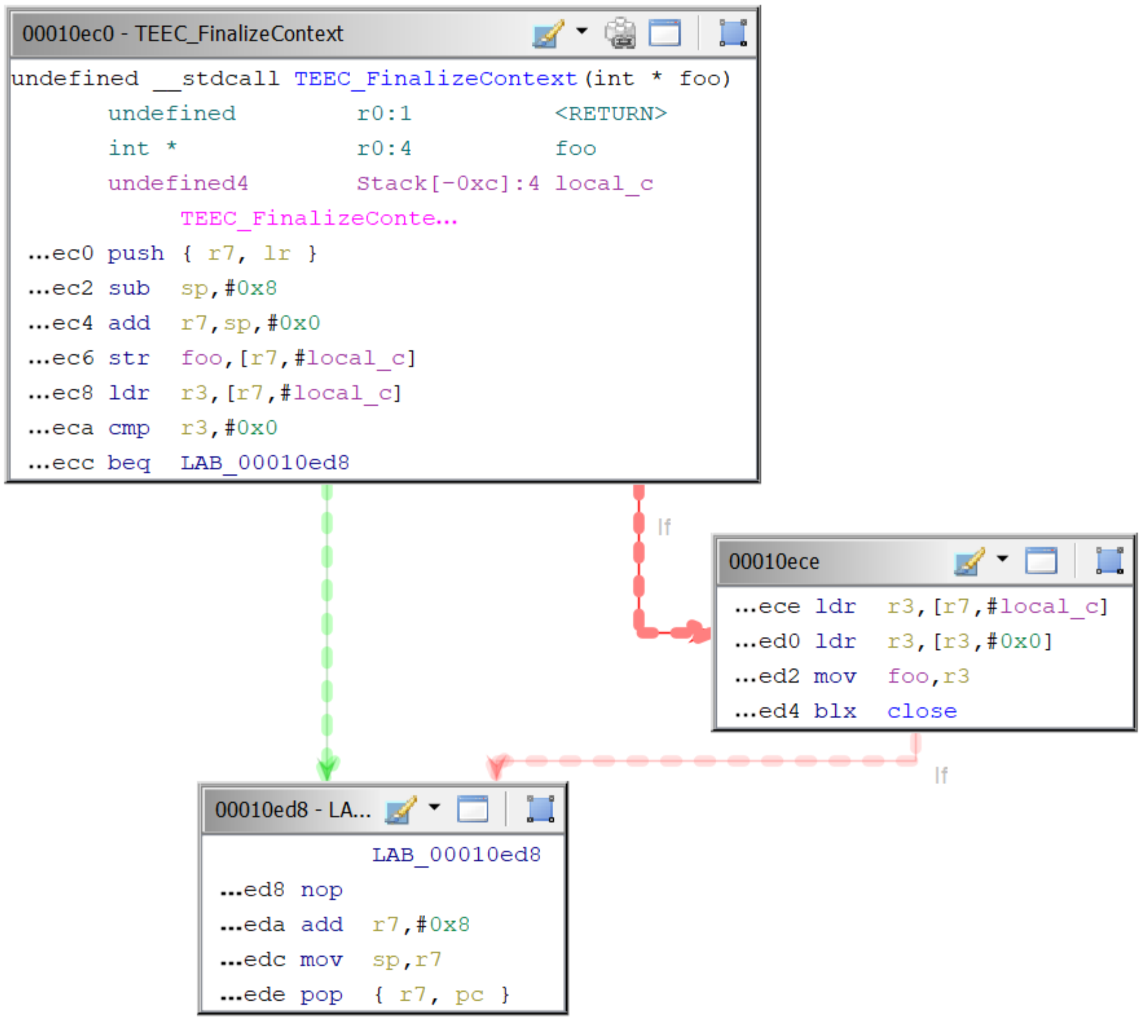
When looking at a binary, it can be impossible to know where execution may end up inside of a binary. This thesis will not explore techniques for discovering subroutines within a binary that are not directly accessed in some form. Binaries can have subroutines that call subroutines through complex sets of instructions that are impossible to determine the exact bounds. This causes problems with the approach we take in inferring type on individual registers at call-time, where we can't reasonably decide what would be inside a register. This is a problem that is not explored by this thesis. Instead, we estimate what may be in a register by noting when a register is derived from something that is an input to a register.

4.6 Control Flow Graph

A control flow graph is a directed graph which represents the control flow through a function. A *path* is a set of nodes and vertices connecting them. Execution of a control flow graph involve starting at the root node, where the subroutine begins, and walking the graph until the node you visit has no outgoing edges. Figure 4.1 shows a simple control flow graph with no *cycles*. A cycle is a set of nodes and edges which, starting at any node in the graph, you can visit each other node contained in the set an infinite number of times by traveling along an edge in the aforementioned set.

4.7 P-Code

P-Code is described as, "a *register transfer language* designed for reverse engineering applications"[14]. This language is used as an intermediate representation to be consumed by the decompiler, which allows for Ghidra to be processor agnostic with regards to its ability to lift code from assembly to C syntax. Operating on p-code allows for the analysis engine to (theoretically) support any architecture that can be lifted into p-code. This thesis



```
undefined __stdcall TEEC_FinalizeContext(int * foo)
        undefined          r0:1           <RETURN>
        int *              r0:4           foo
        undefined4         Stack[-0xc]:4  local_c
              TEEC_FinalizeConte...
 ...ec0 push { r7, lr }
 ...ec2 sub   sp,#0x8
 ...ec4 add   r7,sp,#0x0
 ...ec6 str   foo,[r7,#local_c]
 ...ec8 ldr   r3,[r7,#local_c]
 ...eca cmp   r3,#0x0
 ...ecc beq   LAB_00010ed8
```

```
 ...ece ldr   r3,[r7,#local_c]
 ...ed0 ldr   r3,[r3,#0x0]
 ...ed2 mov   foo,r3
 ...ed4 blx   close
```

```
                LAB_00010ed8
 ...ed8 nop
 ...eda add   r7,#0x8
 ...edc mov   sp,r7
 ...ede pop   { r7, pc }
```

Figure 4.1: A control flow graph for function within a Trustzone Driver

focuses solely on ARM.

Methodology

Combining techniques and applying them to the disassembly of programs trusted to perform communication we can estimate what the type of the argument may be in a somewhat accurate form without needing to actually execute the code within the binary. Observing a binary while executing can give great information for answering questions a researcher may have about a binary. Unless you can exercise all possible execution states for an unknown binary, you won't receive complete information about a binary from simply observing it while it is executing.

This section will outline a type inference scheme based on inferring type on its usage. For example, data that is first dereferenced and then incremented may be an array of data or a data structure, but is very unlikely to be any data that isn't backed by memory. The code written to test this hypothesis was written to work with Ghidra, the NSA's software reverse engineering framework. First the thesis will discuss constructing a control flow graph using Ghidra's API. Afterwards, the thesis will discuss how the control flow graph is iterated and state is stored on an instruction by instruction basis.

Additionally there will be a discussion of patterns and limitations of this method. There are two separate patterns tested in this thesis. One pattern aims to estimate size of an input argument and the other pattern aims to correlate structure members to parameter usage within subroutines.

5.1 Control Flow Graph Construction

Ghidra does not provide an elegant way to get a workable control flow graph for lifting as far as the author is aware. Ghidra offers a way to get individual code blocks at an address, what it believes are the bounds of the block, the blocks successors, and the blocks predecessors. Collecting all of this information we are able to translate this into two separate graph interfaces provided by Ghidra in order to get different types of information.

The first, `ghidra.util.graph.DirectedGraph`, allows for us to access the underling codeblock object easily. After getting the block information for the helper class `ghidra.program.model.block.SimpleBlockModel`, we construct a digraph composed of `ghidra.util.graph.Vertex` and `ghidra.util.graph.Edge`. This graph will be used later for processing p-code in order for the function.

The second, `ghidra.graph.jung.JungDirectedGraph` allows for us to efficiently search for paths between two blocks of a program, which will be important for value-set analysis performed on the registers. This graph is composed of similarly named classes (`ghidra.program.model.block.graph.CodeBlockVertex` and `ghidra.program.model.block.graph.CodeBlockEdge`) and used strictly for its ability to provide us information on paths between two blocks. Constructing this graph done after the construction of the `ghidra.util.graph.DirectedGraph` graph class.

5.2 Control Flow Graph Parsing

After the construction of these objects, we design a high level analysis class, named `Analysis`, to independently walk each p-code operation. This abstraction allows us to combine several types of analysis, if we choose to. There are separate abstractions for the act of walking the p-code representation of each block and actually performing the analysis of how each p-code register is used.

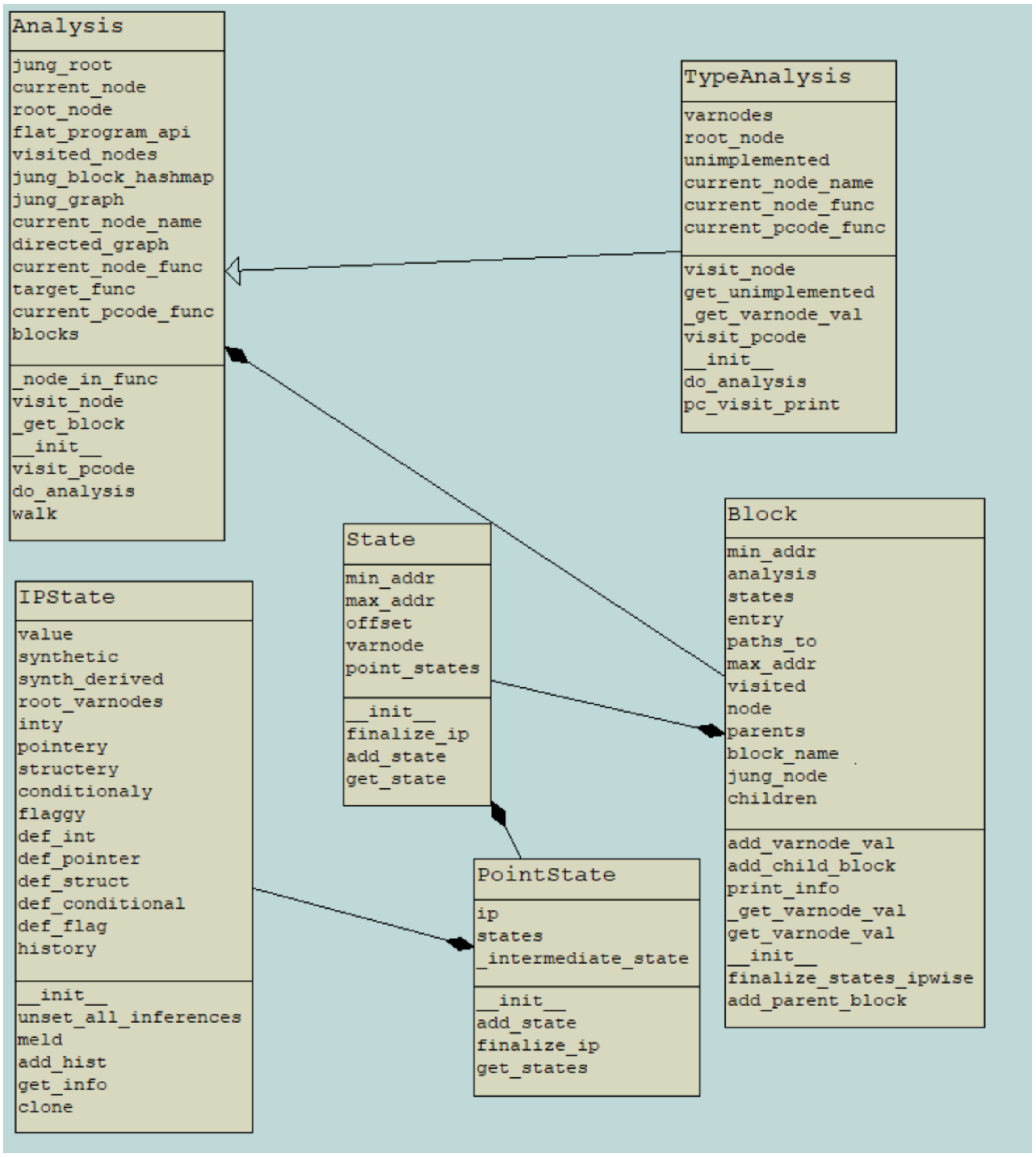
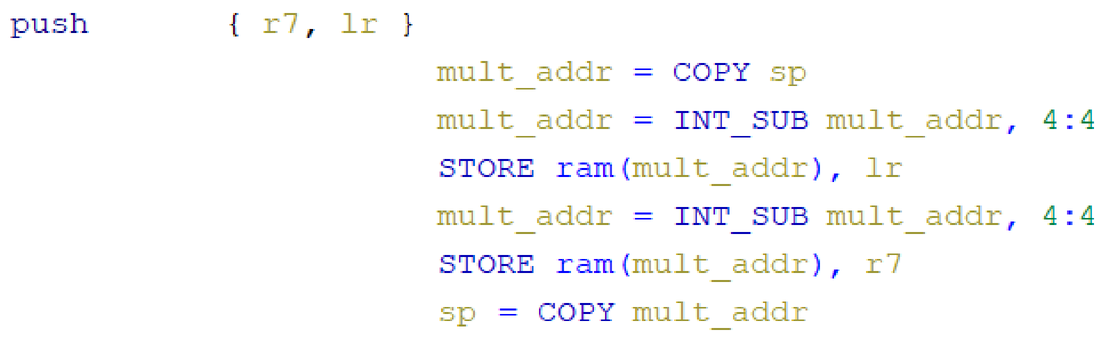


Figure 5.1: UML Diagram describing class layout for type analysis bookkeeping.



```
push        { r7, lr }
                          mult_addr = COPY sp
                          mult_addr = INT_SUB mult_addr, 4:4
                          STORE ram(mult_addr), lr
                          mult_addr = INT_SUB mult_addr, 4:4
                          STORE ram(mult_addr), r7
                          sp = COPY mult_addr
```

Figure 5.2: Instruction for pushing registers onto stack with corresponding p-code.

Block Class Overview

We first begin by constructing a block representation class for each block of code within the function with the assistance of the two directed graph classes. These blocks contain: a reference to the analysis object, the block's children, the block's parents, if itself is an entry node, the Jung node representation of itself, all paths in the graph that lead to the block, the bounds for said block, and state for individual registers within the block. See figure 5.1 for a UML representation of this object. All blocks are instantiated when analysis of the function begins.

When processing p-code the state is saved to the Block object. Each Block keeps a representation of the state for each register which is modified within its bounds. Each `IPState` object is immutable after instantiation to give us the properties of SSA. If multiple states can be present within a block, the phi value is held within a `PointState` object.

In order to understand what a state might be at some point we store each state as it transitions between instructions. This must be done because each p-code operation may access the same register several times within a single non-lifted instruction. Under the current scheme used in this thesis, an operation like figure 5.2 would cause two states for the stack pointer to be generated for a single instruction, which is clearly incorrect. To correct for this, we keep track of the most recent transition made within an instruction and only commit the state after all p-code operations have been iterated.

Accessing state during analysis for a single instruction can be computationally complex but offers the most information at the point in which it occurs. In order to access state, we first search for modifications made for the block we are currently operating in. This means, in the optimistic case, we find a state within the current block. In the pessimistic case, we need to travel to the root node and traverse all paths between the target node and the root node of the function. This operation is equivalent to a forward slice on a particular register with respect to the path. This operation gives us a set of all possible values our analysis thinks can be assumed by this register. Figure 5.1 shows the relationship between `PointState` objects, `IPState` objects, and `State` objects.

5.3 Semantic Size Extraction

While processing p-code, each `IPState` is updated with what it may be according to the operations performed on that register. As data is manipulated there are concrete values that you can infer by adding a symbolic value to keep track of state at a later time. You can also keep track of the history of a value and attempt to understand what several instructions are doing with a single piece of data to accomplish a higher level concept.

For example, when a subroutine is entered there often is a prologue that will save values. See figure 5.3 for an example. Some of these values may be inputs to the subroutine and the address to resume execution at the subroutine's conclusion. This operation directly translates to manipulating data in the running program in order to obtain a result. This is referred to as a stack frame[70]. In a way, stack frames are like anonymous structures that are used by subroutines for bookkeeping. When values are stored in the stack frame, that typically means they are inputs to a function and are required by the subroutine to perform whatever high level operations were translated to the processor's language.

These values, when retrieved, can reveal information about the inputs to the function

```
stmdbsp!,{ r11 lr }
add  r11,sp,#0x4
sub  sp,sp,#0x8
str  param_1,[r11,#local_c]
```

Figure 5.3: Prologue for a function that takes one argument in ARM.

based on how its used. If a value is dereferenced and then offset and used as a function pointer it may indicate that the argument is an object and had its virtual function table[71] which has had a member function called.

By taking each operation and using it to try and derive semantics from the assembly I try to estimate the size of structures and memory areas by assuming accesses are going to be within bounds of the structure at least once. This can help infer the minimum size of a structure used in a function.

5.4 Semantic Type Correlation

Any code that is written will have some amount of code reuse. It can be difficult as a developer to understand when to turn a pattern into its own function. When trying to understand how a complex function works it may help to see how that subroutine is used in other functions. This relation can become even more important when several systems use the same subroutine. Observing behavior that is oblique to whatever interface you are trying to understand can be helpful.

If we are trying to determine what data types compose an argument to a subroutine we can learn more about these functions if we look at sub-subroutines. If sections of other the types we are interested in are used elsewhere, this can give us more data to correctly determine the type of the sub-subroutine and better-estimate the callee's types.

It can generally be said that the arguments to a subroutine have a concrete type when written, but not necessarily when executed. That is to say, as long as the argument conforms to the expected type of the subroutine, the subroutine can fulfill its contract. Consider figure

```c
void access_two(int foo[2])
{
    printf("first foo: %d\n", foo[0]);
    printf("second foo: %d\n", foo[1]);
}
```

Figure 5.4: a function that contractually takes an array of two integers.

```c
void access_one_and_two(int* foo)
{
    printf("first foo: %d\n", foo[0]);
    access_two(&foo[1]);
}
```

Figure 5.5: a function that contractually takes an array of integers.

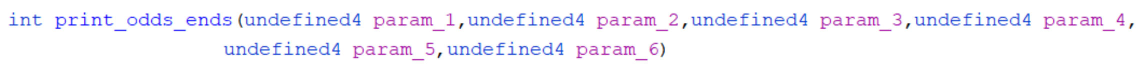
5.4 and what it means to the C compiler. When this subroutine is used, you can pass a pointer to an array of *three* integers and the compiler emits no warnings.

Obviously, looking at the routine as a black box it would be apparent that the code takes an array of two integers. Understanding code in figure 5.4 could help you understand what the code in figure 5.5. Clearly code in figure 5.5 expects an array of 3 integers despite not explicitly declaring the number of members it expects.

Utilizing subroutines that call other subroutines, along with the size estimation semantic we have described, we can use function calls to infer relationships that we wouldn't have normally been able to infer. By using the restrictions placed on the callee's datatype we can more accurately recover the type of a subroutine and successfully correlate several callee data types.

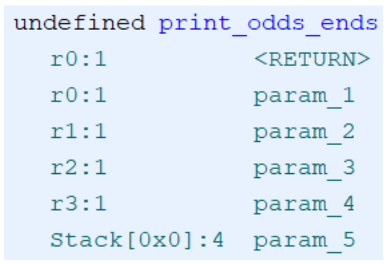
5.5 Limitations

There are a few drawbacks to be aware of when utilizing this form of automated analysis. This section will outline limitations that I encountered when writing this proof of concept tool using Ghidra. There are limitations presented by this approach and the tool



```
int print_odds_ends(undefined4 param_1,undefined4 param_2,undefined4 param_3,undefined4 param_4,
                    undefined4 param_5,undefined4 param_6)
```

Figure 5.6: Inference for a function produced by decompilation.



```
undefined print_odds_ends
   r0:1           <RETURN>
   r0:1           param_1
   r1:1           param_2
   r2:1           param_3
   r3:1           param_4
   Stack[0x0]:4   param_5
```

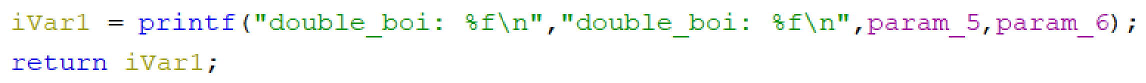
Figure 5.7: Inference for a function produced by disassembly.

that could further reduce the accuracy of results.

This approach makes an assumption about architecture specific semantics that may not necessarily reflect how the high level implementation looks. For example, if the processor demands 4 byte alignment the minimum size of an access to any memory location is at least 4 bytes. This can also be complicated further by odd sized structures, as some compilers will make an assumption that the number of bytes accessible to any allocated memory will be valid up to the minimum alignment size.

This further bleeds into architectural semantics when support for 64 bit registers is not supported and a single argument can span several registers. This affects results of the methodology and can be seen in the Testing and Results section.

Arguments to functions in Ghidra influence how the tool sees what arguments a function takes. We don't try to infer the calling convention of a function and simply rely on how Ghidra would interpret those arguments to a function. If Ghidra hasn't been told what arguments a function takes, it will try to infer what those arguments may be through dis-



```
iVar1 = printf("double_boi: %f\n","double_boi: %f\n",param_5,param_6);
return iVar1;
```

Figure 5.8: Inferred return result for function in 5.6 and 5.7

assembly and decompilation. 5.6 shows a function that has been decompiled and has a reasonably looking signature. 5.7 shows the same function's inferred type with a clear type mismatch between number of arguments and output return type.

Furthermore, Ghidra can make some outright incorrect assumptions sometimes that would be detrimental to fully trusting this tool to produce completely accurate results. 5.8 shows the end of a decompiled function that has the return from `printf` as the actual return value. While functions that return other subroutine results are common, it is exceedingly unlikely that you would check the return of `printf` in this case. This is made even more difficult by the fact that the way this tool gets arguments to a function is by asking Ghidra what it thinks the arguments are. When asking Ghidra for arguments it will get what the disassembly states are the arguments, and not the decompilation.

Additionally, arguments to variadic functions do not give the arguments at the call site but the minimum arguments to a function. This isn't ideal for capturing usage throughout a subroutine, as it requires us to understand the calling convention for an architecture and break cross compatibility gained from analyzing the p-code of a binary.

The approach we take in this thesis has very computationally complex requirements. Some optimizations have been made in order to compensate for harder to solve constraints. For example, we do not consider paths that would visit an already visited node. This allows us to avoid path explosion but also doesn't accurately represent the compiled code. In the future, solving constraints of a looping condition and then producing the correct number of paths would be the most ideal solution.

Testing and Results

This section contains testing data for the outlined approaches in the methodology section of this thesis. The results are derived from the usage of the Ghidra Reverse Engineering Framework[13] and the ideas outlined in the Methodology section to extract the size of an input parameter to a function without directly emulating it. The test corpus consisted of several functions for copying structures of varying size. Additionally, each function had the decompiler's assumed parameters accepted.

The Results for size analysis subsection has a table with the inference code ran to try and assume the size of input parameters for the functions listed within the table. The functions prepended with `copy` are simple copy functions where one result is copied to another argument. The `multi` functions have supplied source code, but are intended to generate multiple code blocks.

The Results for type correlation section discusses the results of the idea discussed in the Semantic Type Correlation section. Functions prepended with `analyze` are functions that accept different structures. Each structure has a shared member. In some instances the code can detect when it has been used as a pointer, but at the moment is not detecting pointer to pointer types.

The table at 6.2 shows test results for the same size inference test ran on 6.1. The table at 6.3 shows a table of type masks for the inferred structure and where the code believes the subtype belongs within the inferred structure.

6.1 Results for Size Inference

Table 6.1: Results for size analysis

Results of Function Inference			
Function Name	Structure Size	Inferred Size	Correct
copy_single_int	4	4	Yes
copy_two_ints	8	8	Yes
copy_single_two_int_structs	12	12	Yes
copy_several_ptr_structs_int	8	8	Yes
copy_single_double	4	8	No
copy_two_doubles	12	16	No
copy_single_two_double_structs	24	20	No
copy_several_ptr_structs_double	8	8	Yes
multi_block_usage_int	8	8	Yes
multi_block_usage_double	16	12	No

6.2 Resources for Result Interpretation of Size Inference

Listing 6.1: Supplemental header for size analysis

```

typedef struct single_int_struct
{
    int first;
} single_int;

typedef struct two_ints_struct
{
    int first;
    int second;
} two_ints;

typedef struct two_structs_struct_int
{
    single_int first;
    two_ints second;
} two_structs_int;
```

```

typedef struct several_ptrs_struct_int
{
    single_int* first;
    two_ints* second;
} several_ptrs_int;

typedef struct single_double_struct
{
    double first;
} single_double;

typedef struct two_doubles_struct
{
    double first;
    double second;
} two_doubles;

typedef struct two_structs_struct_double
{
    single_double first;
    two_doubles second;
} two_structs_double;

typedef struct several_ptrs_struct_double
{
    single_double* first;
    two_doubles* second;
} several_ptrs_double;

void copy_single_int(single_int* src, single_int* dst);
void copy_two_ints(two_ints* src, two_ints* dst);
void copy_single_two_int_structs(two_structs_int* src,\
two_structs_int* dst);
void copy_several_ptr_structs_int(several_ptrs_int* src,\
several_ptrs_int* dst);

void copy_single_double(single_double* src, single_double* dst);
void copy_two_doubles(two_doubles* src, two_doubles* dst);
void copy_single_two_double_structs(two_structs_double* src, \
two_structs_double* dst);
void copy_several_ptr_structs_double(several_ptrs_double* src,\
several_ptrs_double* dst);

void multi_block_usage_int(two_ints *foo);
void multi_block_usage_double(two_doubles *foo);

void multi_block_usage_int(two_ints *foo)
{
    if(foo->first < 50)
    {
        foo->first += 50;
        if(foo->second > 50)
        {
                foo->second -= 50;
            }
        }else
        {
            foo->first -= 50;
            if(foo->second > 50)
            {
                foo->second += 50;
            }
        }
    }
}
void multi_block_usage_double(two_doubles *foo)
{
    if(foo->first < 50)
    {
        foo->first += 50;
        if(foo->second > 50)
        {
            foo->second -= 50;
        }
        }else
        {
            foo->first -= 50;
            if(foo->second > 50)
            {
                foo->second += 50;
            }
    }
}
```

6.3 Results for Type Correlation

Table 6.2: Results for type correlation

Results of Argument Type Inference Size			
Function Name	Structure Size	Inferred Size	Correct
analyze_two_ints	8	8	Yes
analyze_one_struct_member	12	12	Yes
analyze_two_struct_member	12	12	Yes
analyze_three_struct_member	16	16	Yes
analyze_one_struct_member_ptr	8	8	Yes
analyze_two_struct_member_ptr	8	8	Yes

Table 6.3: Results for type masking

Results of Argument Type Inference Masking			
Function Name	Structure Size	Structure Mask	Correct
analyze_two_ints	8	11111111	Yes
analyze_one_struct_member	12	111111110000	Yes
analyze_two_struct_member	12	000011111111	Yes
analyze_three_struct_member	16	0000000011111111	Yes
analyze_one_struct_member_ptr	8	11111111	No
analyze_two_struct_member_ptr	8	11111111	No

6.4 Resources for Type Correlation

Listing 6.2: Supplemental Type correlation header file

```
typedef struct two_ints_inner{
    int first_member;
    int second_member;
} two_ints_struct;

typedef struct struct_one_inner{
    two_ints_struct first_member;
    int second_member;
} one_struct_member;

typedef struct struct_two_inner{
    int first_member;
    two_ints_struct second_member;
} two_struct_member;

typedef struct struct_three_inner{
    int first_member;
    int second_member;
    two_ints_struct third_member;
} three_struct_member;

// for testing the pointer variant
typedef struct struct_one_ptr_inner{
    two_ints_struct* first_member;
    int second_member;
} one_struct_member_ptr;

```

```
typedef struct struct_two_ptr_inner{
    int first_member;
    two_ints_struct* second_member;
} two_struct_member_ptr;


void analyze_two_ints(two_ints_struct* foo)
{
    if(foo->first_member > 5)
    {
        printf("foo->first_member > 5\n");
    }
    if(foo->second_member < 5)
    {
        printf("foo->second_member < 5\n");
    }
}

void analyze_one_struct_member(one_struct_member* foo)
{
    analyze_two_ints(&(foo->first_member));
    if(foo->second_member < 5)
    {
        printf("foo->second_member < 5!\n");
    }
}

void analyze_two_struct_member(two_struct_member* foo)
{
    if(foo->first_member > 10)
    {
        printf("foo->first_member > 10\n");
    }
    analyze_two_ints(&(foo->second_member));
}

void analyze_three_struct_member(three_struct_member* foo)
{
    if(foo->first_member > 10)
    {
        printf("foo->first_member > 10\n");
    }
    if(foo->second_member < 5)
    {
        printf("foo->second_member < 5!\n");
    }
    analyze_two_ints(&(foo->third_member));
}

void analyze_one_struct_member_ptr(one_struct_member_ptr* foo)
{
    if(foo->first_member != 0 )
    {
        analyze_two_ints((foo->first_member));
    }
    if(foo->second_member < 5)
    {
        printf("foo->second_member < 5!\n");
    }
}

void analyze_two_struct_member_ptr(two_struct_member_ptr* foo)
{
    if(foo->first_member > 10)
    {
        printf("foo->first_member > 10\n");
    }
    if(foo->second_member != 0)
    {
        analyze_two_ints((foo->second_member));
    }
}
```

Conclusion

This methodology is effective at capturing the semantics provided to it without program emulation. Encoding program behavior for target architecture can effectively capture data that is interesting to reverse engineers and reduce the amount of noise required to understand a set of problems. When beginning to understand how a system works you may only have a goal in mind.

Reducing the amount of comprehension required to solve whatever challenge you face is optimal for nearly any field of research or task at hand. The approach outlined in this thesis can effectively encode trivial knowledge that can be automated with little work. This automation would reduce overhead work and allow people tasked with understanding how a binary operates to focus on harder challenges than get tasked with something that is trivially apparent to observers.

The work necessary to encode new semantics in this system could be non-trivial. For example, better associating structures between function calls and determining which (estimated) types belong to which functions are a difficult problem. This could eventually be solved by inferring types of members for each structure through usage within the binary. While this thesis explores a very trivial example of this idea, there are limitations imposed on the tool (correctly determining all functions within a binary) and tracking several layers of pointer data types.

Sometimes you cannot count on instrumentation of code to produce results that are helpful. Utilizing this approach allows for the binary to be analyzed without the need for actually executing the underlying binary. This can be helpful for understanding malicious executables and for inexpensive research for hardware you do not own.

Bibliography

[1] Intel. *Intel® 64 and IA-32 ArchitecturesSoftware Developer's Manual.*

[2] ARM. *ARM® Cortex®-A Series.*

[3] Matt Hoekstra. Intel® sgx for dummies (intel® sgx design objectives).

[4] ltd ARM. Introducing arm trustzone.

[5] Cisco Networking. © 2018cisco and/or its affiliates.all rights reserved. this document is cisco public information.page 1of 46white papercisco global cloud index:forecast and methodology,2016–2021.

[6] Ericsson. Ericsson mobility report.

[7] Karl Bode. What the hell is that device, and is it spying on you? this app might have the answer.

[8] Wikipedia. Mirai (malware).

[9] Chris Welch. Sonos explains why it bricks old devices with 'recycle mode'.

[10] BBC News. Apple fined for slowing down old iphones.

[11] Wikipedia. Trusted platform module.

[12] Daniel Gruss Michael Schwarz, Samuel Weiser. Practical enclave malware with intel sgx.

[13] National Security Agency. Ghidra.

[14] National Security Agency. *P-Code Reference Manual*.

[15] Xenofon Koutsoukos henkai Zhang. Generic value-set analysis on low-level code.

[16] Rolf Rolles, editor. *The Case for Semantics-Based Methods in Reverse Engineering*.

[17] Edward J. Schwartz et al. Using logic programming to recover c++ classesand methods from compiled executables.

[18] Cornelius Aschermann Thorsten Holz Tim Blazytko, Moritz Contag. Syntia: Synthesizing the semantics of obfuscated code.

[19] Andrew Ruef. Codereason.

[20] Valgrind. Writing a new valgrind tool.

[21] Shoshitaishvili et al. Intermediate representation.

[22] Intel. *Intel® 64 and IA-32 ArchitecturesSoftware Developer's Manual*.

[23] Intel. Minimum steps necessary to boot an intel architecture platform.

[24] Intel. *Intel 80386 Programmer's Reference Manual*.

[25] Carnegie Mellon University. Virtual memory.

[26] P. A. Karger and A. J. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *IEEE Symposium on Security and Privacy*. IEEE, 1984.

[27] Intel. *Intel® Platform Innovation Framework for EFI System Management ModeCore Interface Specification*.

[28] Shawn Embleton Sherri Sparks and Cliff Zou. Smm rootkits: A new breed of os independent malware. *ACM*, 2020.

[29] Intel. Frequently asked questions for the intel® management engine verification utility.

[30] Intel. Intel virtualization technology(intel vt).

[31] Intel. Intel® software guard extensions tutorial series: Part 1, intel® sgx foundation.

[32] Intel. Intel® trusted execution technology (intel® txt) overview.

[33] Surenthar Selvaraj. Life cycleof an sgx enclave.

[34] Intel. Intel® enhanced privacy id (epid) security technology.

[35] ARM. Arm security technology building a secure system using trustzone technology.

[36] Matthew Gretton-Dann. Introducing 2017's extensions to the arm architecture.

[37] ARM. *ARM® Cortex®-A Series*.

[38] *ARMv8-A Architecture Overview*.

[39] ARM. Trustzone technology for armv8-a.

[40] ARM. *Arm® Cortex®-A73 MPCore Processor*.

[41] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:58, 2016.

[42] Shay Gueron. A memory encryption engine suitable for general purpose processors.

[43] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:62, 2016.

[44] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:3, 41, 2016.

[45] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:62, 2016.

[46] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:62, 2016.

[47] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:85, 2016.

[48] J. Camenisch E. Brickell and L. Chen. Direct anonymous attestation. *ACM*, 2004.

[49] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:85, 2016.

[50] Intel. Enclave interface functions (ecalls).

[51] Intel. *Intel® Software Guard Extensions: Data Center Attestation PrimitivesInstallation Guide*.

[52] *Intel(R) SGX Reference Launch Enclave*.

[53] Intel. *Intel® SGX PCK Certificate and Certificate Revocation List ProfileSpecification*.

[54] Intel. The quoting enclave.

[55] John M. Code sample: Intel® software guard extensions remote attestation end-to-end example.

[56] S. Ayoun J. Sakkien, S. Siddha and S. Katz-zamir. Intel sgx instructions in enclave initialization.

[57] Intel. *Overview of Intel®Software Guard Extensions Instructionsand Data Structure*.

[58] Intel. Intel®software guard extensions (intel® sgx) architecture for oversubscription of secure memory in a virtualized environment.

[59] ARM. Exception model.

[60] ARM. *ARM® Cortex®-A Series*.

[61] ARM. *ARM® Cortex®-A Series*.

[62] arm. *Architecture Reference Manual for ARMv8-A*.

[63] Mike Pritchard. *Devfs – device file system*.

[64] ARM. 2.7. supervisor calls (svc).

[65] ARM. Arm compiler armasm reference guide.

[66] ARM. 2.12.13. secure monitor call (smc).

[67] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams.

[68] Barry Rosen; Mark N. Wegman; F. Kenneth Zadeck. Global value numbers and redundant computations.

[69] Thomas Lengauer; Tarjan; Robert Endre. A fast algorithm for finding dominators in a flowgraph.

[70] Paul Krzyzanowski. Stack frames: A really quick explanation of stack frames and jframe pointers.

[71] Microsoft. Virtual function tables.