CipherTrace: Automatic detection of ciphers from execution traces

Mostafa Abdelmoez Hassanin

Kellogg College
University of Oxford

A dissertation submitted for the
MSc in Software and Systems Security

Abstract

Cryptography is defensive in nature, as it provides confidentiality, integrity, and authenticity to its consumers for data processing, storage and transmission. The employment of cryptography in software has introduced additional complexity in investigating the intent of a binary program or a running process. The rapid-increasing trend of using cryptography in malicious software [54], has re-introduced cryptography from an "active-defense" perspective. This trend is known as cryptovirology, which was first introduced in 1996 [68]. The ever-increasing number of malware samples employing cryptography has demanded for more sophisticated analysis techniques to account for the malware and its operation(s), and if applicable, allow for neutralization and/or recovery. Generally, the detection and identification of cryptographic functions, routines, or algorithms facilitates malware analysis and forensics [66, 67]. However, most of the attempts presented in the literature have basically followed a top-down approach, which (by design) has limited them, as they are mainly focused on detecting certain implementations, rather than the building blocks, elements, or ingredients of cryptography in general. Nevertheless, the lack of work on automating the process makes it harder to scale in quantity and quality. In this dissertation, we explore the idea of generalizing the detection of cryptography, via dynamically analyzing Out-of-VM execution traces of binary programs. The prototype we developed (`CipherTrace`) is based on PANDA, the architecture-neutral dynamic analysis platform [18]. PANDA was favored among others mainly due to its plugin architecture as well as its record-and-replay functionality, which allows for developing extensions easily and repeatable analysis. PANDA is based on QEMU, which is a whole-system emulator and virtualizer. In QEMU, the in-guest components are not visible to the guest programs, therefore it provides isolation to a certain extend, wherein the results are more true-hearted. We apply custom heuristics on the synthetic information of called functions, e.g., count of arithmetic operations, number of executions of basic blocks, memory access patterns, and more. The synthetic information can be categorized as raw (i.e., Platform-driven or Assembly-driven), or lifted (i.e., LLVM-driven). Finally, we report the findings, observations, limitations, and assert the feasibility and scalability of the concept, the dynamic analysis approach, as well as the technique applied to identify cryptographic (crypto) elements to classify a cipher. **Crypto elements** are the generic elementary units (or constituents) of cryptography, i.e., the operational functional blocks (e.g., a substitution or permutation step, etc).

Acknowledgements

I would like to pay my special regards to Ivan Martinovic, my supervisor, for his valuable advice and support. Also, I would like to extend this regards to Alessanda Cavarra, the director of graduate studies, for her valuable input and feedback. I am also indebted to my wife and my family for their unconditional support. And I wish to thank all the people whose assistance was a milestone in the completion of this dissertation. And at the time of writing, I cannot help but notice my laptop's fan noise and heat–screaming for recognition–and since I got this far with it, I highly appreciate that it hasn't failed me–yet.

Contents

List of Tables

List of Figures

List of Algorithms

1 Introduction

1.1 Motivation

As of 2017, ”every 4.2 seconds a new malware specimen emerges” [57], reporting an increase of 35% in 2020 [58]. As of 2018, ”1 in every 13 web requests lead to malware” [55], and since then, the average cost of a malware attack on a company is estimated to be more than $2.6 million [1]. In 2020, it was estimated that around 30% of breaches involve malware [10], and the average cost of a data breach on a business is estimated to be more than $3.92 million [30]. Moreover, 23% of malware incidents involve ransomware [10], and the cost to recover from a ransomware has skyrocketed to 84,000$ in 2020–which projects a 104% increase from 2019 [12]. It is predicted that cybersecurity spending are to hit 1$ trillion in 2021 [60].

In the last decade cryptography has been weaponized in authoring malware, moving from encrypting binaries, code blocks and botnet communications, to giving birth to a new class of malware: ransomware. A ransomware is essentially a malware based on cryptography [54]. The trend of using cryptography in malicious software has been rapidly emerging, especially in the last 5 years [54], and that re-introduces cryptography from an ”active-defense” perspective. This trend is known as cryptovirology, which was first introduced in 1996 [68].

Malware analysis is an essential process to understand the behavior of a malware, and to allow for neutralization and/or recovery. In which, Reverse Engineering (RE) plays an integral part in discovering the internal principles of code. Static analysis approaches have been challenged for decades, whereas despite the fewer challenges that dynamic analysis approaches face, they are more rewarding as they provide insights in the runtime behavior of a binary [42]. ”There are various techniques employed by malware authors to evade detection, e.g., code obfuscation, dynamic code loading, encryption, and packing” [42]. ”Dynamic analysis tools are robust to such techniques, but yet imperfect” [42], and there is no silver-bullet tool that rules-them-all, and that becomes an order of magnitude harder when cryptography is employed by malware authors, specifically when it comes to detecting cryptographic (crypto) algorithms and their keys.

A large volume of new malware samples are discovered daily. ”Even worse, malware is rapidly evolving, and becoming more sophisticated and evasive” [66], and that results in billions of dollars being poured into fighting malware, and millions of dollars are being poured into neutralization or recovery from malware. The ever-increasing number of malware samples employing cryptography has demanded for more sophisticated analysis techniques to account for the malware and its operations. The prospect of making 23% of malware incidents obsolete is quite rewarding [10], as any ransomware out there becomes obsolete if the crypto key(s) have been extracted in runtime, after detecting and identifying the crypto operation.

Therefore, fighting malware (and specifically speaking ransomware) is one of the direct use-cases of this dissertation. In a recent survey on the detection techniques for ransomware,

Berrueta et al. analyzed 63 ransomware samples from 52 families–including the most spread ones up until 2019 [3]. They propose a collection of algorithms aimed to detect and classify ransomware activity (but not recovery) and they reviewed 37 related papers. Actually, around 75% of their ransomware sample utilized AES encryption [3], which is an algorithm that our prototype could detect and classify as a `Fixed S-Box` block cipher, and even extract the `key`, the `ciphertext`, and the `plaintext` from the memory. The same applies to the `AES` algorithm used in the mainstream OpenSSL library, which is used to handle HTTPS connections. Additionally, our prototype was experimented on the `TwoFish` algorithm used in KeePass (the password manager software), in which the algorithm was classified as a `Pre-computed S-Box` block cipher. Generally, there are many other use-cases in which our proposed tool will help a great deal in malware analysis and digital forensics. For instance, since it is able to extract the `plaintext`, so the encrypted code blocks in binaries (i.e., in malware) could be recovered in runtime, not to mention the HTTPS communications could be bared.

Generically speaking, our task is to detect and identify crypto operations and derive a classification for the ciphers employed. In the "Background and Related work" section below, we provide an introduction to the various domains that this dissertation addresses. Then, we follow up by an in-detail survey of the related work. We highlight that, most of the literature has essentially followed a top-down approach in detecting and identifying crypto algorithms and has not touched upon key identification or extraction. Moreover, the literature is mainly focused on identifying certain crypto algorithms or implementations, rather than looking for the crypto elements used to construct them, i.e., the operational functional blocks of cryptography, which poses a limitation on the related work. Nevertheless, the lack of work on automating the process makes it harder to scale in quantity and quality. In the following section, we present our contribution, and as this dissertation continues we demonstrate how hard the problem is.

1.2 Overview

In this section, we outline the structure of this dissertation, which implicitly profiles our methodology. Our dissertation consist of three parts: research, Application, and reflection. Each part may consist of multiple phases.

1.2.1 Research

We conduct an initial research phase which is resembled in the Background review of analysis approaches, obfuscation, and crypto identification techniques. We conclude this phase of research with a Proof of Concept, which is considered a feasibility study. In which PANDA (the platform we use in `CipherTrace`) was used in a similar domain to determine if crypto keys are being ex-filtrated outside the network [19], and also to decrypt a song protected by DRM [62]. In the second phase of research, we delve into the depth of more than 8 tools from the Related work, and how do they perform the instrumentation, feature extraction and identification stages of various crypto algorithms' implementations.

1.2.2 Application

We lay out our Scope, as well as the Threat Model which we adhere to. In the Application part, we mainly focus on applying our concept and testing its feasibility, wherein the choices

and decisions aim for simplicity and to highlight our contribution. Nevertheless, the Application part methodologically consists of two main phases: implementation and evaluation, which include verifying our results to validate our findings. In the implementation phase, the dynamic analysis approach and the crypto algorithm identification technique are manifested in the Design and architecture of our application, featuring the two processing stages, i.e. the engines: the `Inspection Engine` and the `Analysis Engine`. The former is the dynamic analysis platform (PANDA), and the later is the multiple stages of analysis we go through. Both are described in the "System implementation" section below. We implemented, designed and experimented our application (`CipherTrace`), additionally the Source code, test algorithms and test data are available below.

In the Evaluation and results phase, our mandate is that, the results are reproducible and verifiable. We achieved that by following a white-box approach. In this phase, we first relay our Criteria for evaluation, then we evaluate `CipherTrace` against the comparable Related work. In order to fulfill our mandate, we develop a `Verifier` module, which is specifically developed to raise a flag when a discrepancy in our `Analysis Engine` is detected. Add to that, the various reverse engineering techniques employed to validate the output of our `Inspection Engine`, as well as verifying the results of our `Analysis Engine`. Last but not least, we provide a forensic-like Report in the Appendix to record everything we did to allow for reproducible results.

1.2.3 Reflection

Lastly, we conclude and reflect on our journey in the "Conclusion and reflection" section. In addition to, outlaying the potential Future work, i.e., enhancements and/or improvements.

1.3 Concept and thesis

We believe that, the research conducted in the field of automatic detection and identification of cryptography is mostly algorithm-oriented, rather than key-oriented (such as in [34, 36]). Additionally, and most importantly, it is focused on detecting the implementations of a certain crypto algorithm (e.g., AES, RC4, etc.), rather than classifying a cipher (e.g., block cipher, stream cipher, etc.). In other words, most of the research to-date follows a top-down approach, which makes it quite limited to the algorithms or the implementations that it has been experimented with. Whereas, we aim to take a bottom-up approach, leveraging higher-level semantics by employing PANDA, since usually dynamic analysis approaches lack such a level of granularity [52]. We also apply a different technique in detecting and identifying the crypto algorithms, in which we initiate our analysis by looking for the operational functional blocks of cryptography (e.g., substitution or permutation steps), then end up by deriving a classification of a cipher.

> We assert if the following thesis is feasible and rewarding: following a bottom-up approach in detecting cryptography, and to derive a classification of the utilized cipher–via employing Out-of-VM dynamic analysis approach. In other words, towards the generalization of crypto primitive detection and cipher identification. A scenario of our thesis is outlined in the "Application" section below.

Cryptography is a routine-based paradigm. In other words, it relies on the repeated execution of pre-and-well-defined steps (or instructions), i.e., routines. In most of the cases (if not all)

these instructions involve a high number of arithmetic operations, and they correlate with memory access patterns usually in ratio to the size of the `plaintext` or `key`. Moreover, routines translate into loops in application code, and loops are exhibited as jumps in machine code, e.g., Assembly. On an instruction level, a given set of instructions constitute a basic block of code [11]. A basic block of code is basically the set of instructions that has a single entry and a single exist, as explained in "Basic Blocks (TransExec cycle)" section below. In `CipherTrace`, we associate the basic block(s) of code with a higher-level semantic (e.g., a function) while attempting to avoid a potential semantic gap limitation.
Then by employing Information Measurement tests on function input/output, we attempt to locate the crypto functions' candidates. Then we reduce the search space by applying some heuristics based on the extracted Synthetic Information. Some of the heuristics we came up with, hence, the "custom" notion. After reducing the search space, we delve deeper into the analysis to detect crypto elements, by correlating the candidates with "static" heuristics such as loops, no of executions, etc. Then we apply "dynamic" heuristics (e.g., memory access patterns) to identify the crypto element, and we correlate that with other crypto elements as well as with the notion of a memory block to denote a block cipher. Finally, we perform Memory Re-construction to verify our findings. The crypto detection and identification criteria are discussed in the "Classification of crypto algorithms" section below.

1.4 Scope

1.4.1 Overview

Our project is based on security analysis. The appropriate security analysis has to fulfill three rules: stealthiness, isolation and no increased attack surface [31]. Our dynamic analysis approach (`VMI--Emulation`) fulfills isolation and to the best of our knowledge, it does not introduce a new attack surface. Refer to Table 1.1 for an overview of the approaches mentioned in this dissertation, against the three rules of security analysis.

Approach	Stealthiness	Isolation	No new attack surface
DBI	No	No	Unknown
DBI w/ MPK	No [59]	No	Unknown
Processor Tracing [8]	No	Yes	Unknown
VMI–Driver	No	Yes	Yes
VMI–Emulation	No	Yes	Yes
VMI–VT	Yes	Yes	Yes
HW-VT	Yes	Yes	Yes

Table 1.1: Overview of dynamic analysis approaches from a security standpoint

Our scope is focused on detecting and identifying encryption algorithm(s) in product ciphers, specifically in block ciphers. We do not aim to evaluate the security implications of the dynamic analysis approach, rather than the feasibility of applying the concept and evaluating it via `CipherTrace`. We take code and data obfuscation out of scope, but not distorting the binary or function names in code. This leaves us to the threat of hiding crypto activity; in which we focus on detecting and identifying any potential crypto activity, wherein we take false-positives out of scope.

1.4.2 Threat Model

Intro

A threat model usually outlines what we are worried about, and the assumptions that can be checked or challenged. Additionally, it may survey the potential threats, along with the actions to address each one of them. `CipherTrace` is considered an action to the threat of hiding crypto activity. Moreover, validating our model is manifested in our ”Background and Related work” section below. Last but not least, evaluating the actions taken to address the mentioned threats is essentially our ”Evaluation and results” section.

We also outline an adversary’s profile to model a realistic scenario. We believe that, a threat landscape will be incomplete without a profile. A profile highlights the intentions and capabilities of an adversary which is highly likely to reflect the adversary’s decisions along the course of an attack.

Landscape

In dynamic analysis, the threat landscape relays a wide attack surface. The attack vectors can be categorized as conceptual, environmental and technical. In our threat model, we limit our attack vectors to conceptual and technical only. For example, the framework or platform used for dynamic analysis is considered an environmental attack vector, thus out of scope.

Model

As far as the security of the platform is concerned, we are not seeking stealthiness, as it is extremely hard to achieve, and it is a research area of its own. Rather we are seeking isolation and no increased attack surface. PANDA[18] (the platform we selected) fulfills both to the best of our knowledge. Our main goal is to select an appropriate platform wherein our Concept and thesis can be applied relatively easily with as much isolation as possible, so that the results are more true-hearted. In the ”Conclusion and reflection” section, we dive deeper into that decision.

The adversary’s profile is a seasoned cyber criminal, but possesses limited knowledge in cryptography. The adversary is well-aware that, re-inventing cryptography is highly likely to result in a weak algorithm. Weak crypto is easy to attack, and could jeopardize the whole operation.

As a result, the adversary invests in distorting well-known cryptography algorithms’ implementations, without affecting the functionality. We assume the adversary is going to pick a well-known implementation of a cryptographic algorithm and adapt it–changing its details, e.g., key size, function names, etc. However, it will remain conceptually and functionally the same. The adversary employs the implementation into the binary program with the objective of it being undetected, or not correctly identified, or being extremely hard to recover the crypto `key` from. We also assume that, the adversary’s binary is being executed on a Windows platform.

2 Background and Related work

2.1 Background

2.1.1 Binary Analysis

In 2008, the BitBlaze project [52] implemented a hybrid binary analysis platform that combines static and dynamic approaches to incorporate the advantages and compensate for the disadvantages. One of the highlighted challenges was lack of higher-level semantics in dynamic binary analysis, which are present in source code, e.g., functions, memory buffers and data types [52]. On one hand, dynamic analysis offers insights into the behavior of the executable in runtime. On the other hand, static analysis provides static or signature-based insights, and explores all potential execution paths without executing the program. The latter has been challenged for decades by obfuscation, and additionally the accuracy of the results is heavily challenged by potential morphism [42]. In `CipherTrace`, a dynamic analysis approach will be taken, because we are more interested into exploring the real execution path and analyzing the actual dynamic state of the binary or the process by analyzing various state changes in the execution environment.

2.1.2 Dynamic Binary Instrumentation/Introspection

Binary Instrumentation is the technique that inserts extra code into an arbitrary program to collect runtime information. Despite its inappropriateness for security-related tasks such as binary analysis, Dynamic Binary Instrumentation (DBI) remains commonly used for analysis, hardening and bug hunting [15, 31]. Due to overhead, Just-In-Time (JIT) compiler noise and other factors, an application can detect whether it is running in the context of a DBI framework, and it could even escape the instrumentation engine–with some limitations and prerequisites [31]. In the case of using Intel PIN's framework, DBI could even introduce additional attack surface, since Intel PIN has issues with self modifying code [31]. The most fundamental problem is that, DBI logic and the application reside in the same address space, with no isolation present. Intel Memory Protection Keys is a possible mitigation [31], such as in [44, 59]. Static Binary Instrumentation (SBI), Virtual Machine Introspection (VMI) and Hardware Virtualization are better alternatives [15]. However, SBI inherits the challenges and limitations of static analysis. Hardware Virtualization is stealthier, offers better performance, and more stable than VMI-based techniques [43], but it's not yet fully-fledged [47] in terms of consumerization (available tools), compatibility, and interoperability [16, 17, 41, 43]. To date, perhaps DRAKVUF [32] is the most commonly used binary analysis system that is VMI-based (virtualization), and leverages hardware virtualization extensions, but it only runs on Intel CPUs, and it doesn't offer memory dump API–even though it is excellent for real-time memory analysis. When it comes to emulation in VMI-based techniques, as part of the BitBlaze project, TEMU [52] was introduced, which is a VMI-based (emulation) dynamic binary analysis framework. We argue that TEMU is an In-VM approach because it requires a

driver to be installed in the guest Operating System (OS) for instrumentation; notably, it only supports Windows XP as guest OS [52]. In 2015, PEMU was introduced [69], which is an Out-of-VM dynamic binary instrumentation framework. TEMU and PEMU leverage QEMU, which is a Virtual Machine Monitor (VMM) (or Emulator) and Virtualizer. In [69], a brief comparison between similar existing platforms to-date was presented, placing PEMU as the most fit compared to its rivals as far as instrumentation level, API-support, isolation, OS compatibility and interoperability are concerned. PEMU has no available documentation, and it is no longer supported. PANDA [18] is also QEMU-based, and it fulfills PEMU's features and functionalities, except it lacks on Intel PIN's interoperability–and for that, PEMU offered limited support. Moreover, PANDA offers architecture neutrality and modular design. This makes it generic and developer friendly, despite its limitations and performance overhead when it comes to tainting and LLVM support–which is adequate to our purposes. Therefore, we believe that it is the most appropriate choice. Also, it offers record-and-replay functionality and it has a very strong taint analysis capability. In `CipherTrace`, we will use PANDA for introspection. Refer to Figure 2.1 for an overview of the mentioned Out-of-VM frameworks and platforms.

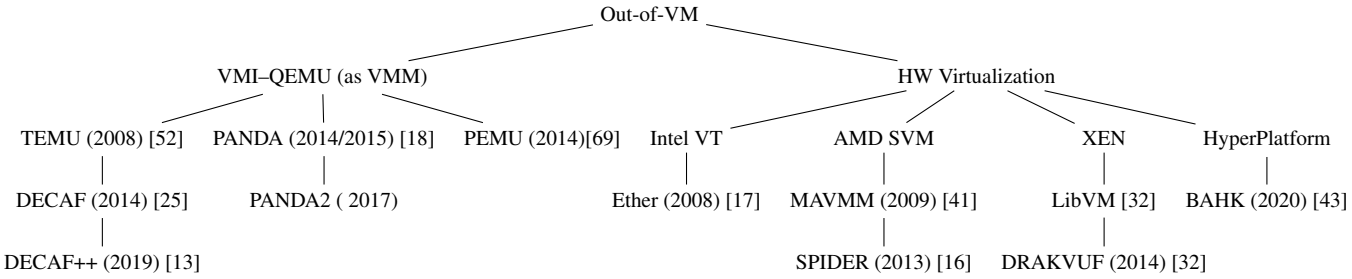

Figure 2.1: Overview of different Out-of-VM platforms or frameworks

2.1.3 Obfuscation

An obfuscation scheme is a binary program's transformation to evade reverse engineering and analysis without changing the functionality. We categorize obfuscation schemes to binary, code and data obfuscation. Binary obfuscation aims to obfuscate the binary itself, mainly to evade static analysis techniques, e.g., packing [42]. Code obfuscation aims to obfuscate the control flow or code blocks in a program to decrease the readability, e.g., flattening the control flow, or obfuscating the loops [29]. Data obfuscation aims to obfuscate the data flow or memory management in a program [29].

All the Related work mentioned in this dissertation adopt a DBI approach, therefore, they are resilient to code packing as only the executed code will be instrumented. `CipherTrace` is also resilient to code packing for the same reason. Tools such as `K-Hunt` and `bacs` additionally consider Virtual Machine (VM) obfuscation, anti-virtualization and various anti-debugging tricks [33, 34], which is out of our scope. Apart from `Aligot`, `CryptoHunt` and `K-Hunt`, all the other tools do not consider code or data obfuscation. `Aligot` authors claim that their tool is able to detect unrolled loops [7]. `CryptoHunt` is able to deal with data obfuscation by combining loop I/O relations with bit-precise symbolic execution [65]. Last but not least, `K-Hunt` is able to deal with non-standard key buffers [34].

2.1.4 Identification of crypto implementations

To the best of our knowledge, the first two papers that have had the focal point of identifying the implementations of crypto algorithms in binaries (following a dynamic analysis approach) are [36, 63]. Lutz [36] developed a tool that automatically decrypts network traffic in the task of revealing encrypted botnet communications. Wang et al. [63] addressed the problem of revealing cryptographic algorithms during execution in the task of automatic protocol reverse engineering. Both and many others such as [7, 14, 24, 26, 28, 33, 34, 65] applied a Dynamic Binary Instrumentation (DBI) approach. Moreover, they relied on Intel PIN, the DBI framework that is only compatible with Intel processors [35], except for [36]. Lutz [36] used Valgrind [40] for easier taint-tracking and monitoring of memory access patterns. We note that, Valgrind is not compatible with Windows [36]. All the aforementioned tools are heuristics-based, except for [26, 28] are Machine Learning (ML) based, and they are trained on known implementations. Moreover, [28] is not available as a tool to trial. Refer to Table 2.1 for an overview of the aforementioned papers. We add `CipherTrace` for comparison.

Name	**Framework**	**Technique**	**Task**
Lutz (2008)	Valgrind	DBI-Heuristics	Decrypting network traffic
Gröbert (2010)	Intel PIN	DBI-Heuristics	Identification of Cryptographic Primitives
Aligot (2012)	Intel PIN	DBI-Heuristics	Cryptographic function identification in obfuscated binary programs
Hosfelt (2015)	Intel PIN	DBI-ML	Classification of cryptographic algorithms
Kali (2017)	Intel PIN	DBI-Heuristics	Address Aligot's scalability issues
CryptoHunt (2017)	Intel PIN	DBI-Heuristics	Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping
K-Hunt (2018)	Intel PIN	DBI-Heuristics	Hunt Insecure keys
CryptoKnight (2018)	Intel PIN	DBI-ML	Classification of cryptographic algorithms
BACS (2018)	Intel PIN	DBI-Heuristics	Identification of cryptographic algorithms
CipherTrace–This dissertation	PANDA	VMI-Heuristics	Detection and identification of ciphers

Table 2.1: Overview of crypto identification techniques

2.2 Related work

At the time of writing, the most recent and closest work to our concept but in a different context or setting, and for a different purpose is [56]. In this work, given a `ciphertext`, and employing Machine Learning (ML), the author explored the idea of identifying block ciphers of five different algorithms: AES, DES, 3DES, RC5 and Blowfish, with an accuracy of 90%.

As far as detecting and identifying the implementations of crypto algorithms, section 2.1.4 above presented an overview of the related work, along with the frameworks employed. We dive a bit deeper into the technique(s) employed by the related work to detect and identify crypto algorithms or their specific implementations.

Crypto primitive detection relies mostly on instrumenting 'jump', 'call' and 'return' instructions, in addition to memory access patterns. Any project that tackles that, employs instrumentation/introspection and tainting (in-process, whole-system, or bit-level) which goes in-tandem with memory analysis (static or live). The related work employs instrumentation, in-process tainting, and static memory analysis. Generally, such a project goes through a few stages, and they are;

- Instrumentation: obtaining an execution trace–refer to the ”Framework” in Table 2.1.
- Features Extraction:
  - Basic blocks: For instance, by detecting ’jump’ or ’return’ instructions such as in [24, 26, 34], and/or relying on the DBI framework as specified in [33].
  - Loops; and they are detected as follows:
    * Backwards edges in a Control Flow Graph (CFG) [36].
    * Repeated execution of the same code address such as in [7, 24], which is similar to [26, 34, 65].
    * Signature-based lookup, employing a Data Flow Graph (DFG) such as in [33].
  - Entropy measurement (Shannon’s): Mostly by measuring the entropy of memory buffers to find candidate functions such as in [24], which employed memory tainting similar to [26, 34, 36].
  - Crypto constants: Either by employing them to a certain degree [7, 24], or being heavily dependent on them [33].
  - Arithmetic Operations:
    * On instruction level, i.e., from ’operands’ and based on ’opcodes’ [26, 34, 65]. For example, ’`add r1, r2`’, or ’`xor r1, r2`’, etc.
  - Buffer mingling: In which a crypto basic block is meant to mingle the data and key memory buffers, exhibiting certain characteristics to look for [34], which is conceptually similar to [7].
  - Correlations: For example, between the execution length and input size to identify a candidate basic block or key [34], which is similar to [7, 65].
- Identification: identify the algorithm or decrease the search space such as in [24], in which CFG was employed to reconstruct the memory to find semantic relationships. Worth to mention that in [7] loops were associated with the data flow (employing a DFG) similar to [33].
  - Heuristics: Such as in [7, 14, 24, 26, 28, 34], which are derived from standard algorithms or even certain implementations. They can be categorized to dynamic heuristics (i.e. based on function I/O) such as in [7, 34], or static heuristics such as the rest of the related work discussed in this dissertation. In [33], basic heuristics are employed to manually select a segment of the trace to run the signature-based constant identification sub-system, followed by the mode-of-operation identification sub-system (employing a DFG).
  - Symbolic Mapping: Such as in [65], in which the loop body is transformed to symbolic formulas to capture bit-level semantics of crypto transformations, which is resilient to control and data obfuscation at a bit-level [65]. However, it is heavily based on reference implementations.
  - Machine Learning: Such as in [26, 28], and both are trained on the implementations of standard algorithms, and [26] employs the key size as a feature, and utilizes *constants* to train the model.

In section ”Classification of crypto algorithms” below, we comparatively relate our analysis to the stages above.

The closest work to `CipherTrace` to-date is SysTaint [61]. SysTaint is a PANDA-based taint tracking system for malware analysis. SysTaint allows for detecting cryptographic functions based on function-level custom heuristics. It doesn't differentiate between encryption, compression, or encoding functions. The objective is to taint their output for better data flow tracking on their inputs and outputs to facilitate an analyst's task.

SysTaint collects the following information: number of basic blocks per function, number of executed basic blocks, loops (inferred from the previous two), number of arithmetic and total instructions, memory buffer size, entropy test (Shannon's) and the number of printable characters (ASCII) in each memory buffer. It applies the following heuristics: read/write high entropy data, shallowness of root node in a call tree, and the presence of loops. At least two must be fulfilled to mark a function as cryptohraphic to decrease false positives, e.g., 'memcpy' OS function call [61].

SysTaint is developed for a different purpose, and applies different techniques from `CipherTrace`, and it has some limitations, most notably:

- It employs callstack analysis to identify higher level callers. Due to the semantic gap and its callstack analysis, in-lining a cryptographic function in the body of a larger function hinders its heuristics.

SysTaint is implemented as a PANDA fork (with changes), in addition to a few PANDA plugins, most notably (and related to our work) are "fn_memlogger" and "fn_composition" plugins. The plugins gather and report information about the called functions in a given PANDA-replay trace.

In the assessment section of SysTaint below, we assess it comparatively to `CipherTrace`.

3 Application

3.1 Overview

We complement our Concept and thesis by the following scenario:

> Given a malware sample or an arbitrary binary program, after running the binary through `CipherTrace`, Synthetic Information about the binary will be extracted, e.g., called functions, number of times a function got executed, number of basic blocks in a function, count of arithmetic operations, memory access patterns and many more. So that, after analyzing the extracted information, insights on the employment of cryptography in the program will be produced.

Our implementation consists of the following core components;

- `CipherTrace Inspection Engine`: PANDA2. In which, a PANDA plugin will be developed leveraging its API callbacks to intercept instructions and memory accesses. Then, collect, process and extract the information after potential processing to accomplish a sufficient level of semantics. PANDA can provide higher-level semantics utilizing its APIs, e.g., function calls. Refer to the ”New PANDA plugin for CipherTrace” section below for more information.
- `Randometer` Module: A program that processes the extracted information from the previous process, and pinpoints the `callers` of the functions to be further explored based on Information Measurement criteria. This module is part of the `Analysis Engine`.
- `Analyzer` Module: This is the main module in the `Analysis Engine`, which performs information processing to analyze the output of the `Randometer` module based on algorithms discussed in section ”Classification of crypto algorithms” below.
- `Reporter` Sub-module: A program that reports the findings during the analysis process. This sub-module is part of the `Analyzer` module of the `Analysis Engine`.
- `Verifier (Default)` Module: A program to perform Memory Re-construction to verify the reported findings to increase the confidence level in `CipherTrace` results. This module is part of the `Analysis Engine`.
- `Verifier (Light/Heavy)` Sub-modules: They are considered part of the `Verifier` Module. The ”light” version reports if the ”default” version is expected to find any ”data series” (e.g., `key`, `plaintext`, or `ciphertext`), and the ”heavy” version performs arranged Memory Re-construction over all interesting points in the execution (i.e., memory locations) looking for the ”data series”.

Our implementation also consists of the following complementary modules in the `Analysis Engine`;

- `Tester` Module: This program tests the core analysis algorithm (with test data) to detect and identify crypto elements.
- `Visualizer` Module: this program takes a `graph` file as an output from the `Analyzer` Module and plots it as a graph for visualization and also for validation of the results and analysis.

In summary, our implementation involves two processes; inspection and analysis. In the inspection process, we rely on PANDA (our plugin). In the analysis process, we sequentially execute the different `python` modules that we developed. The `Verifier` module, and its de-facto sub-modules (Default, Light, and Heavy) are also part of the `Analysis Engine`, i.e., the analysis process. When unspecified, the `Verifier` refers to the ”default” version.

Refer to the CipherTrace Manual in the Appendix for the operational details of `CipherTrace` engines.

3.2 Concept, approach and technique

Given our Motivation the objective is to detect the presence of a product cipher, which is the cipher that combines substitution, permutation and modular arithmetic operations [21], wherein we plan to experiment on AES. Our bottom-up approach can be summarized as follows:

- Search for a S-Box: a substitution step.
- Search for a P-Box: a permutation step.
- The presence of S-Box and P-Box indicate a Substitution Permutation Network (Product Cipher).
- Correlate with memory access patterns, e.g., buffer size and number of accesses to search for a ”block” notion (i.e., a block cipher).

In AES block cipher and many others, ”ShiftRows” and ”MixColumns” work in tandem. By shifting rows, every byte of the `ciphertext` depends on every byte of the `plaintext`. By mixing columns, every column of the `plaintext` does not affect the corresponding one in the `ciphertext` [21]. In other words, that is considered as a permutation step all-together.

An overview of most of traditional ciphers in-use today inferred from [21] can be categorized as follows:

- Traditional Cipher
  - Product Cipher: S-box and P-box, and modular arithmetic
    - Block Cipher
      - Substitution–permutation network, e.g., AES
      - Feistel Cipher, e.g., DES, Blowfish
    - Stream Cipher, e.g., RC4

- Substitution Cipher
  - Mono-alphabetic Cipher
    - ROT13: a Caesar Cipher
  - Poly-alphabetic Cipher
    - Auto-Key, Vigenere, Playfair, Hill Cipher, One time pad, Rotor Cipher
- Transposition Cipher
  - Columnar Transposition Cipher
  - Rail-Fence Cipher

To summarize, in the task of applying the described concept, and as mentioned in sections 2.1.1 and 2.1.2 above, we follow a dynamic analysis approach characterized by the PANDA platform. This approach further evolves to manifest itself with a `VMI-Heuristics` technique when it comes to identifying crypto elements (and ciphers) based on heuristics–as outlined in section 2.1.4 above. Therefore, as part of the System implementation, we describe our use of PANDA in the "Inspection Engine (Platform)" section below. In that section, we also describe the Analysis Engine (Analyzer), which handles the identification of the crypto elements in a multi-stage analysis. By design, since we are adopting a bottom-up approach (i.e., a low-level concept) we expect that `CipherTrace` will be resilient to some Control Flow obfuscation. For example, a block cipher has to have a substitution step (i.e., S-Box) which cannot be evaded on the level we analyze at.

3.3 Design and architecture

We adopt a modular design in `CipherTrace`, so that modules can be executed in a standalone mode. Additionally making `CipherTrace` easily scaled, improved, or even re-designed. As mentioned earlier, `CipherTrace` consists of two processes: `Replay Inspection` and `Stats Analysis`. They resemble the two main components in `CipherTrace`, which are the `Inspection Engine` and the `Analysis Engine`. The output of the `Replay Inspection` (which features PANDA) is fed as an input to the `Stats Analysis` process, which may optionally feature PANDA when verifying results. In `Stats Analysis` process, multiple stages of analysis take place (`FILTERING`, `CRYPTO ELEMENTS IDENTIFICATION`, and `REPORTING`). All of which adopt a **modular** design–as defined in section 3.1 above.

The overarching scenario is as follows: We start from a binary program executed in the PANDA environment, and end up with the execution report, graphs, and the interesting points in the execution. The New PANDA plugin for CipherTrace (`func_stats`) outputs '`func_stats`' file. Refer to Figure 3.1 below for an overview of the solution design and architecture highlighting the involvement of PANDA, and featuring what we go through in our engines. In this figure, we aggregate and match PANDA's translation-execution cycle mentioned in section "Basic Blocks (TransExec cycle)" below with our plugin's semantics.

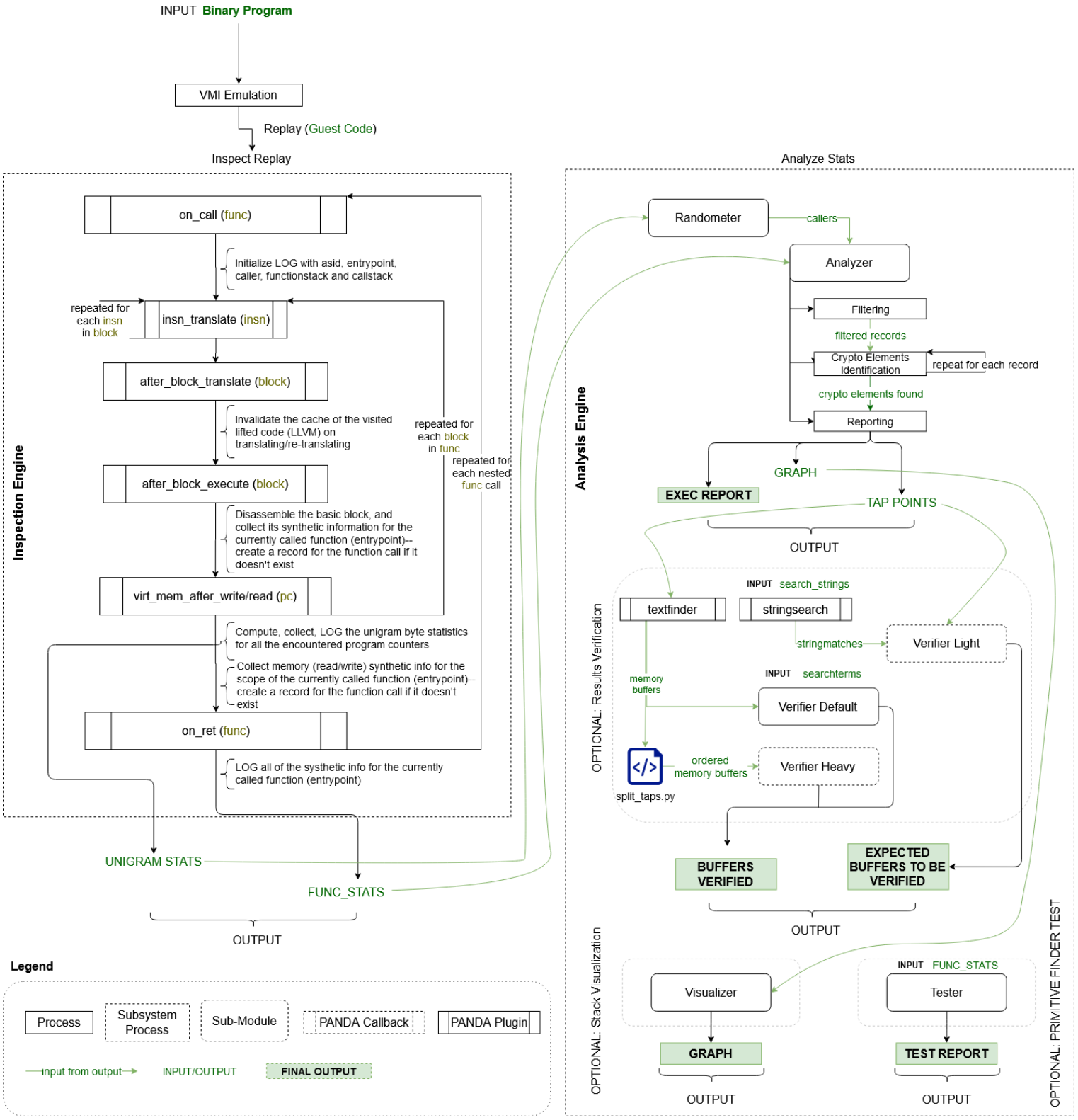

Figure 3.1: Solution Outline

3.4 System implementation

3.4.1 Inspection Engine (Platform)

Basic Blocks (TransExec cycle)

In compiler construction, a basic block (BB) is a code sequence (i.e, set of instructions) with no in/out-branches except for two: one for the entry, and another for the exit [11]. This restriction makes it highly amenable to analysis [11]. PANDA accomplishes emulation via basic block translation. It disassembles the code into guest instructions (one by one) and simultaneously assembles a parallel basic block of instructions in an intermediate language (IL). From this IL, QEMU generates a corresponding basic block of binary code that is directly executable on the host to emulate the guest behavior. QEMU toggles between translating guest code and executing the translated binary versions. Refer to Figure 3.2 for an illustration of the translation-execution cycle, along with the PANDA callbacks (CB) invoked as explained in this video–featuring a co-author of PANDA.

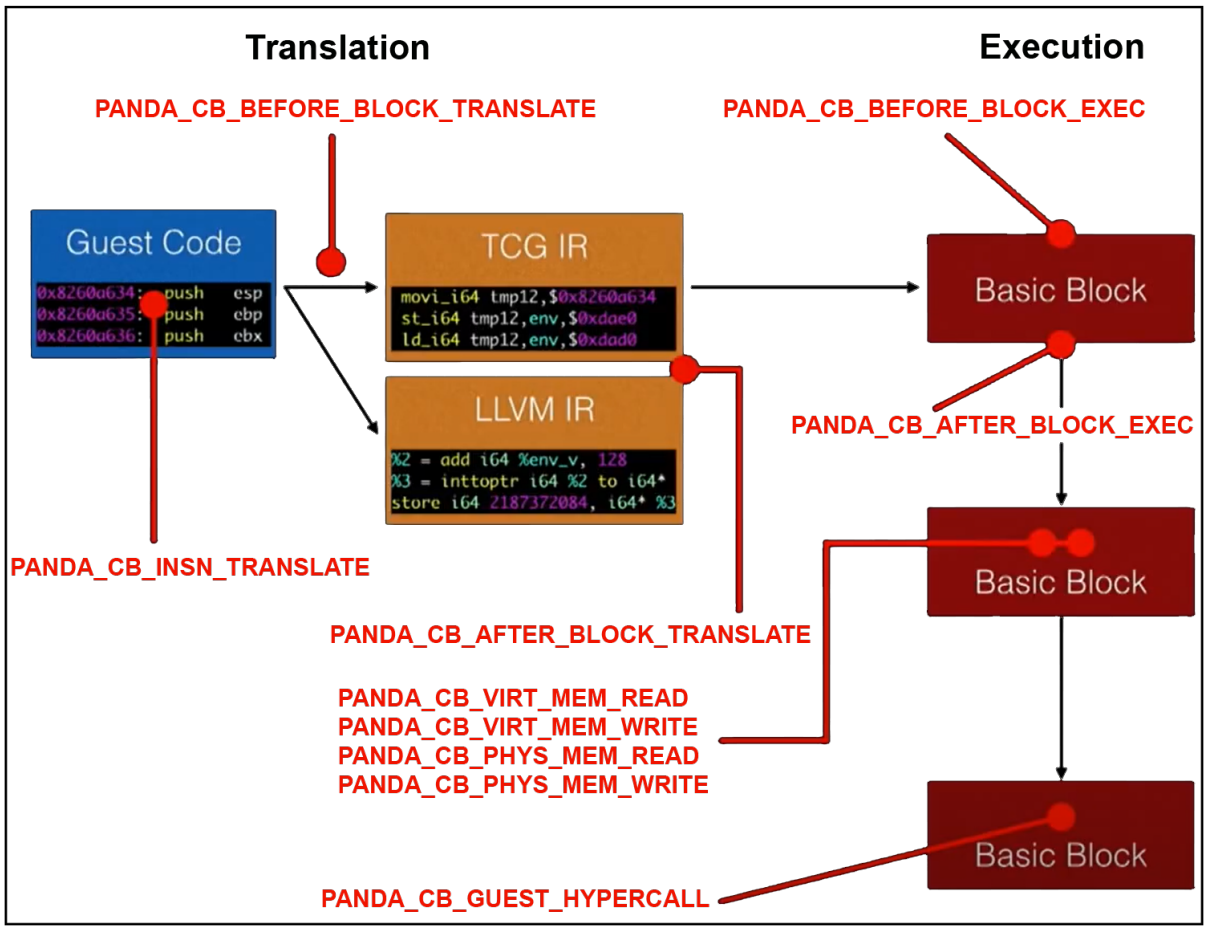

Figure 3.2: PANDA Translate-Execute Cycle Process

Memory Tainting

Memory Tainting is the process of tracking the propagation of flagged data in memory to determine the data flow. Dynamic binary analysis requires a taint-checking technique to instrument/inspect the data flow. PANDA leverages whole-system tainting such as [13, 25, 32]. In which it labels a memory buffer and track it along the exclusion in the so

called ”Dynamic Taint Flow Analysis” process. This process is similar to [13] but slower, and if LLVM is enabled PANDA lifts QEMU’s TCG Intermediate Representation (IR) to LLVM IR. Enabling LLVM allows the analysis to take place in a simplified but semantically equivalent domain, however on the expense of performance [18]. Refer to Figure 3.3 for an overview of PANDA’s taint analysis which is explained in this video–featuring a co-author of PANDA. There are two terms that accompanies a tainting process: a **taint source** and a **taint sink**, denoting where it starts from and where it ends.

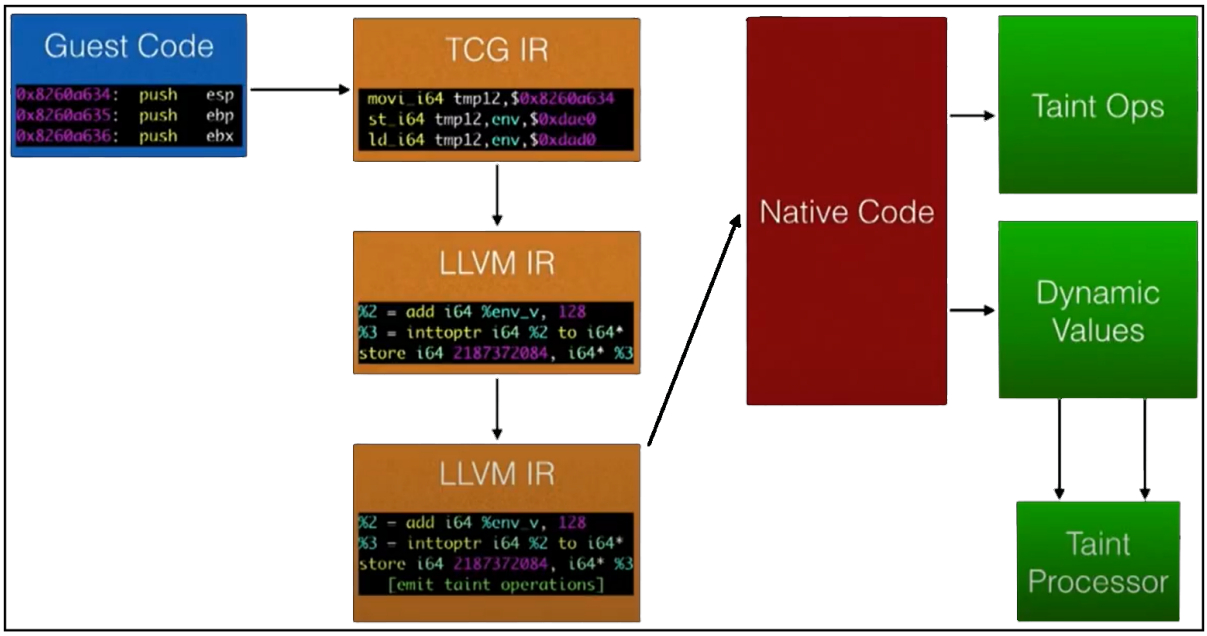

Figure 3.3: PANDA Dynamic Taint Analysis Process

Key concepts

It is quite important to understand a few key concepts about PANDA:

- A tap point is a specific execution state at a certain point, usually a memory state at a program counter. It is a triplet of caller, program counter, and the address space (also known as ASID or the CR3 register).
- When dealing with tap points, a content sample to search for has to be given (e.g., a `stringsearch`). Alternatively, grouping the tap points by distance to a statistical model, or grouping the tap points that handle similar data (i.e., clustering) is necessary. The `bigrams` plugin allows grouping and `correlatetaps` plugin reports the tap points that were often seen to be writing to contiguous memory regions in close temporal proximity.

Limitations

Since in `CipherTrace` we plan to use PANDA, it is worth to mention its limitations–most notably–it is limited to:

- Fifteen levels of callstack information whereby it is not always known in advance how many levels of callstack information are required to find an interesting memory location, i.e., a tap point.
- Fixed `stringsearch` by 1-byte counter and 256-bytes search.

Proof of Concept

In [19] and by using PANDA, Dolan-Gavitt et al. managed to taint a crypto key until it has ended up in a syscall (i.e., a socketcall), denoting that it was ex-filtrated over the network. In another scenario, the authors managed to locate a SSL master key by feeding the `keyfind` plugin a packet capture to start from. The fundamental idea was to locate the code that generates SSL/TLS master secret, and from which it finds potential keys and tests them wherein the expected size was taken as a search criteria. It also had a limitation that direct jumps to addresses won't be detected. Last but not least, in [62] Wang et al. managed to decrypt a song protected by DRM. That was by computing a histogram of memory buffers that go through each function call then pin-point potential functions that does decryption via randomness and entropy tests. Then by searching the function "writes" of `ciphertext` to memory, it dumped the output of the function that handles about the expected size–the output was the song. Note that, sizes were known in advance and were used as a search criteria to decrease the search space, which is not the same as our technique, and not even close to our Concept and thesis. However, that was enough to prove that PANDA is more than adequate to our purpose and it offers the necessary capabilities.

PANDA for CipherTrace

We use the following plugins from PANDA:

- The `asidstory` plugin: to identify the different processes that exist in a replay and the portions of the replay in which they were active. We use it to identify the ASID of the specimen, i.e. the binary program we wish to analyze.
- The `unigrams` plugin: to collect unigram byte statistics (i.e., a histogram of byte values seen) for each tap point encountered in a replay for memory reads and writes. We use that as the main input to the `CypherTrace Analysis Engine`, specifically to the `Randometer` module.
- The `func_stats` plugin is the New PANDA plugin for CipherTrace that we developed. It collects Synthetic Information about the called functions in a replay. We use the output as an input to the `Analyzer` module in the `CipherTrace Analysis Engine`.
- The `textprinter` plugin: dumps the memory buffers at certain tap points. We use its output as an input to the `Verifier` module, in its "default" and "heavy" versions for Memory Re-construction. As mentioned previously, the `Verifier` module and its sub-modules are part of the `CipherTrace Analysis Engine`.
- The `stringsearch` plugin: to dump string matches and their tap points, so that the `Verifier Light` sub-module is able to perform its look-up for expected "data series", i.e., keys and texts.
- The split_taps.py script that is shipped with PANDA is used to arrange the output memory buffers from the `textprinter` plugin and produce "dat" files–each file is for the corresponding tap point–so that the `Verifier Heavy` can perform arranged Memory Re-construction.

New PANDA plugin for CipherTrace

Firstly, the name of our plugin is `func_stats`, and its Source is available below. Secondly, even though our plugin was influenced by `SysTaint`'s two plugins in information collection and logging, our plugin has a different design, implementation, use, as well as strategy. Additionally, our plugin collects more data and applies different technique(s). A notable similarity in relation to Synthetic Information collection, we used the same 'opcodes' determining 'arith' and 'mov' instructions and they are;

- `add,adc,sub,xor,shr,shl,div,mul,rol,ror,dec` for the former.
- `mov, lea` for the latter.

More 'opcodes' could be added, but the ones we listed above have achieved the desired results. The other similarities are irrelevant to mention, because they are far from our contribution and they do not affect the Evaluation and results.
The similarities are either open-source or third-party-library related, or even programming language specific, i.e., some C/C++ coding practices.

The design and architecture of `func_stats` plugin is outlined in the `Replay Inspection` process in the Solution Outline figure above. As per to the purpose of this plugin (i.e., the Synthetic Information which we collect and log) are listed and described in the following section. Additionally, in `func_stats` plugin we run, collect, and log Shannon's entropy test on the memory buffers for each function call–which is discussed in section "Information Measurement" below. Last but not least, the operational Input/Output parameters for this plugin are listed in the CipherTrace Manual in the Appendix.

Synthetic Information

In this section, we list and describe the synthetic information we collect and log as part of our new plugin (`func_stats`) in Table 3.1 below. We categorize the information as Platform-driven (or generic), Assembly-driven (from the dissembled basic blocks), or LLVM-driven (from the lifted LLVM code). The former two are described as 'raw', whereas LLVM-driven information is quite dynamic and verbose, since it relies on the lifted code in the LLVM Intermediate Language (IL). Worth to mention that, the Platform-driven information also relies on the assembled TCG IL. TCG IL is an intermediate language for the QEMU Emulator. The purpose of an IL in our case is to achieve architectural neutrality [3.4.1]. The synthetic information are the output of our plugin (`func_stats`), we call that report a `LOG` that contains records of function calls.

Field	Description	Driven By
instr_count	The guest instruction count at which the record has been collected This is a unique identifier across the whole replay	Platform
asid	The address space identifier	Platform
entrypoint	The called function address, equivalent to the initial program counter	Platform
caller	The caller address, equivalent to the return address of the current stack entry	Platform
callstack	The callers of the current function at hand: 1) equivalent to the return address, i.e., the program counter of the basic block + its size 2) features the blocks that have led to the function call	Platform
functionstack	The function calls themselves, not what has led to them (i.e., not the callstack)	Platform
distinct_blocks	The number of all distinct basic blocks executed	Platform
maxexecs	The highest number a basic block got executed–if duplicate, the first encountered one is returned	Platform
maxexecs_addr	The address of the most executed basic block	Platform
sumexecs	The sum of all executions of all blocks; comparable with 'llvm_bb' field	Platform
nreads	The total bytes read from memory	Platform
reads	The memory "reads" array Each read has a 'base', 'entropy', 'len', 'nulls', 'pc', 'printableChars'. Refer to Table 3.2 for more info.	Platform
nwrites	The total bytes written to memory	Platform
writes	The memory "writes" array Each write has a 'base', 'entropy', 'len', 'nulls', 'pc', 'printableChars'. Refer to Table 3.2 for more info.	Platform
insn_arith	The number of times arithmetic operations were executed	Assembly
insn_movs	The number of times memory operations were executed	Assembly
insn_total	The total number of instructions executed	Assembly
llvm_bb	The number of all visited basic blocks	LLVM
llvm_fn	The number of of all visited functions	LLVM
llvm_modules	The number of all visited modules	LLVM
llvm_insn_alloc	The number of times memory allocation instructions were visited	LLVM
llvm_insn_arit	The number of times binary (or logical) operators were visited	LLVM
llvm_insn_call	The number of times 'call' instructions were visited	LLVM
llvm_insn_intrinsic	The number of times intrinsic instructions (e.g., 'memcpy', 'memmove') were visited	LLVM
llvm_insn_load	The number of times memory 'load' instructions were visited	LLVM
llvm_insn_store	The number of times memory 'store' instructions were visited	LLVM
llvm_insn_tot	The total number of all visited instructions	LLVM

Table 3.1: Description of the synthetic information logged by our plugin

Last but not least, below are the list of synthetic information we log for memory "reads" and "writes" in a function call, i.e., in a LOG's record.

Field	Description
base	The memory address where the memory buffer starts
len	The length of the memory buffer
entropy	Shannon's entropy test for the memory buffer
nulls	The number of nulls in the memory buffer
pc	The program counter of the memory buffer
pritableChars	The number of printable characters in the memory buffer

Table 3.2: Description of memory fields logged by our plugin

The fields listed and described in Tables 3.1 and 3.2 above are employed in the Classification of crypto algorithms–which are discussed later in detail.

3.4.2 Analysis Engine (Analyzer)

Information Measurement

In communication theory, entropy is a concept that was first introduced by Shannon more than fifty years ago [50]. In information theory, entropy is a parametric measure for information diversity (or the de-facto disorder). Assuming that, the only way to get to the most diverse configurations (i.e., equilibrium) is when they are randomly-mixed [45]. Shannon's entropy is probabilistic in nature [50], and it has some short-comings when it comes to the concept of variance, order, and normally distributed data [46]. Our data (i.e., the memory buffers) are considered uni-formally distributed [62], since the outcome of a byte in a "strongly" encrypted data buffer is equally likely. There has been many developments on the concept of entropy since the fifties [22, 45]. As a result, alternative entropy tests can be applied, wherein some are more appropriate for certain purposes. For example, employing a relative entropy test to address Shannon's entropy limitation on normally distributed data [46], which is a limitation that doesn't apply to us since our data are considered uni-formally distributed. However, for the sake of simplicity, and the ease of comparison with the Related work (focusing on the concept), we select Shannon's entropy test for memory buffer measurement in our plugin (`func_stats`), since it has been employed in most (if not all) of the Related work in the same context. We discuss that decision in our "Conclusion and reflection" section below, and discuss improvements as part of the Future work.

The Chi-square randomness test is a statistical hypothesis test, which is designed to test for information distribution [22]. It is mostly used to test the randomness of the pseudo-random number generators [62]. In [22], the authors managed to find a relation between entropy tests (including Shannon's) and Chi-square statistic under certain conditions. The Chi-square test is appropriate to our purpose, since it tests the null-hypothesis [22]. In inferential statistics, the null-hypothesis is a default position wherein there is no relationship between two measured phenomena or no association among groups [22]. Therefore, we decide to adopt that test in the `Randometer` module, in the task of analyzing the called functions' **Input/Output Relation (IOR)** to search for encryption functions. An encryption function is defined as the one that has low-randomness input and high-randomness output. Wang et al. reported positive results when employing the Chi-square test in the same context [62]. We reflect on that decision in the "Conclusion and reflection" section below.

We re-write the `find_drm.py` script [62] as a starting point for our `Randometer` module. We enhance it, add more options as well as logging capabilities. We use it to find encryption functions or routines. In detail, we suppress the entropy measure logic (Shannon's Entropy) as we are no longer looking for compressed mp3 files or encoded data buffers [45]. Additionally, we swap the randomness test (Chi-Square) logic to look for the representation of an encryption algorithm (or function) rather a decryption operation. Finally, we make it highly configurable to match different appetites. In our our plugin (`func_stats`), we use Shannon's entropy to be better aligned with the related work. We use it to collect and log the entropy of memory "reads" and "writes" (i.e., the data buffers) for each function call. Whereas in our analysis, we use Chi-Square randomness test due to the positive results reported when it was used in a similar context.

In the "Limitations and discussion" section below, we discuss the limitations of our various decisions. Additionally, in the "Configurational" and "Technical" sections of the Future work below, we reflect on our decisions proposing potential enhancements and improvements.

Classification of crypto algorithms

In this section, we describe the techniques we apply to detect and classify the crypto algorithms. Comparative to the Related work, we also go through the two *stages* of "Feature Extraction" and "Identification". In the "Feature Extraction" stage, we detect and identify the crypto elements (i.e., the functional blocks of cryptography), so that we could derive a classification for the utilized crypto algorithm in a routine or a function in the "Identification" stage, wherein we apply our custom heuristics. Our classification is considered implicit, because we conjointly survey the identified crypto elements to derive a classification of a traditional cipher–as outlined in section 3.2 above, which will be discussed in-detail in the following sections.

In section 3.4.1 above, we list and describe all the synthetic information we collect and log as part of our plugin which compose our features. We depend on the platform (refer to section 3.4.1) in extracting some features such as identifying the basic blocks, collecting memory access patterns, and LLVM-driven synthetic information. Additionally, similar to [7, 24] and others, we detect loops by looking for the repeated execution of the same basic block. As discussed in the previous section, we also employ Shannon's entropy test to measure the diversity of information in memory buffers as a measure for randomness–similar to [24] and others. In [26, 34, 65], arithmetic operations are extracted on an assembly-level, we additionally adopt LLVM-based binary (or logical) operators which provide more insights as it is also quite verbose and fulfills architecture neutrality. We adopt **Input/Output Relations (IOR)** in the task of searching for candidates of encryption functions–as described in the previous section. We also employ IOR in finding crypto elements which will be discussed later. However, the key size was not taken as a search parameter or criteria in order to be generic as per to our concept, unlike [34] and many others.

For the sake of increasing the readability of this section, we define the following;

- The `state` is represented by the expression $s_{initial}..s_{final}$: starting from the $initial$ state and ending with the $final$ state, whereby $s_{initial}$ is the `plaintext` and s_{final} is the `ciphertext`. It mainly refers to the state of a memory buffer which manifests the internal state of a cipher [21].

- A record is essentially a **function call**. A LOG is a set of all records $r_{0_{field}}..r_{ifield}$, whereby i denotes the index of a record, and $field$ is the name of a field in a record. **Let** R be a set of all the records in a LOG, which is represented by $R = r_0, r_1, r_2, r_n$.
- **Let** C be a set representing the records of a certain $caller$ value in R–given $r_{i_{caller}}$.
- `mainRecs` are the main records for a certain $caller$ value: the records having the highest loop count.
  Let `mainRecs` be a set represented by $M = m_0, m_1, m_2, m_n \longrightarrow$ *main records selection*
- **Let** N be a set that represents $C - M$.
- `recStack` is the "`stack`" field name of a given record. It is an array of `callstack` or `functionstack` represented as $r_{i_{stack_l}}$, whereby `stack` denotes a `stack` name and l denotes the length. The default stack name is "`functionstack`" since it provides the high-level semantics we require.
- `distinctStackRecs` are the distinct records in N for a specific `recStack`. Let that be set N^*.
- For each one of N^*, the `sharedStackRecs` are the records in R that share the same `recStack` $\longrightarrow$ *stack filtering*
- `filteredStackRecs` are the filtered records in `sharedStackRecs` as per to some Information Measurement criteria $\longrightarrow$ *records filtering*

DEFINITION 1: Filtering
Filtering is a three step process, in which we carry out main-records selection, stack filtering, and records filtering. The input is the `LOG`*, and the output is the graph (or CFG), i.e., the* `filteredStackRecs`*. That is according to certain Information Measurement criteria: loop count, number of arithmetic operations and write entropy.*

We start the `FILTERING` stage by feeding the output of the `Randometer` (i.e., the `callers`) to the `Analyzer` module. For each `caller`, we filter out the records that share the same `caller`, then we exclude the records having the highest count of an executed basic block. After such exclusion, we could be as low-level as possible in the control flow with better performance. And if we skip the exclusion, we have more records but it doesn't affect our results. After the exclusion, we carry on with filtering all the distinct records for a specific `recStack` (i.e., stack name). So that, for each one of these records we fetch the matching records from the `LOG`. Lastly, we filter the result for the records which fulfill certain Information Measurement criteria (mentioned in the algorithm definition below), and on which we execute the `CRYPTO ELEMENTS IDENTIFICATION` stage.

The filtering algorithm is as follows;

Algorithm 1: Filter `LOG` records (R)

FILTER (R)
- **inputs :** All `LOG` records `R`
- **output:** The `filteredStackRecs` denoted by F
- `distinctStackRecs` is N^*
- **foreach** *distinctStackRec* $d_i \in N^*$ **do**
  - `f_stack` is `recStack`
  - `f_write_entropy` := 1.0 ;
  - `f_mainrec` := "maxexecs" ;
  - **foreach** *record* $r_i \in R$ **do**
    - **if** $r_{i_{f\_stack}} = d_{i_{f\_stack}}$ **then**
      - tmpRec := r_i;
      - mem_writes := $tmpRec_{writes}$;
      - write_entropy_count := 0;
      - **foreach** *mem_write* $w \in$ *mem_writes* **do**
        - **if** $wentropy >$ `f_write_entropy` **then**
          - write_entropy_count = write_entropy_count + 1;
      - **if** *write_entropy_count* > 1 & $tmpRec_{insn\_arith} > 1$ & $tmpRec_{f\_mainrec} > 1$ **then**
        - $F \xleftarrow{+} tmpRec$
- **return** F;

DEFINITION 2: Crypto Elements
In the `CRYPTO ELEMENTS IDENTIFICATION` *stage, we start from where the previous stage has left off to detect and identify crypto elements for later classification. A crypto element is defined as a functional building block of a crypto function or routine. Therefore, we categorize crypto elements to: state, scheduling, initial key-round, key-round, substitution box (sbox), a permutation step (mixing and/or shifting).*

The main algorithm for detecting and identifying crypto elements is as follows:

Algorithm 2: Find the crypto elements in `LOG` records (R)

```
CRYPTOELEMENTFINDER (F)
    inputs : The filteredStackRecs denoted by F
    output: The results dictionary denoted by PF_OUT
    // Solve() function; groups records by entrypoint and aggregates "maxexecs" of same entrypoint
    aggregatedFilteredStackRecs := Solve(F, 'entrypoint', ['maxexecs', 'aggregated']);
    foreach aggregatedFilteredStackRec ar ∈ F in aggregatedFilteredStackRecs do
        // celement_tofind & prereq_celements are of type enum.
        IdentifyCryptoElement(celement_tofind, prereq_celements, PF_OUT, F, aggregatedFilteredStackRecs):
    return PF_OUT;
IDENTIFYCRYPTOELEMENT (CELEMENT, PREELEMENT, PF_OUT,
 aggregatedFilteredStackRecs, F)
    inputs : The crypto element to find, denoted as CELEMENT.
             The prerequisite elements, denoted as PREELEMENT.
             The global results object to fill, denoted as PF_OUT.
             The aggregatedFilteredStackRecs, F aggregated.
             The filteredStackRecs denoted by F.
    output: None
    // Sorted() function; sorts in desc order, so that we can find the prerequisites first
    sAggregatedFilteredStackRec = Sorted(aggregatedFilteredStackRecs.items(), reverse=True);
    foreach entrypoint ep in sAggregatedFilteredStackRec do
        foreach filteredStackRec fr ∈ F do
            if fr_entrypoint = ep then
                if CELEMENT in PREELEMENT then
                    return return;
                else
                    /* FindPreCElement() function; carries out the standalone searches in the foreach below, but no
                       prerequisite records are passed. It covers scheduling, sbox, mixing                          */
                    FindPreCElement(PREELEMENt)
                // Fill_PFOUT() function; fills the result in the global results object
                foreach p ∈ PREELEMENT do
                    begin
                        switch p do
                            case CElement.initkround do
                                initkround_result := Find_AddKRound_Init(fr, schedulingRecord);
                                if initkround_result['found'] then
                                    Fill_PFOUT('initkround', fr, initkround_result, PF_OUT);
                            case CElement.kround do
                                kround_result := Find_AddKRound(fr, initkroundRecord);
                                if kround_result['found'] then
                                    Fill_PFOUT('kround', fr, kround_result, PF_OUT);
                            case CElement.shifting do
                                shifting_result := Find_Shifting(fr, mixingRecord);
                                if shifting_result['found'] then
                                    Fill_PFOUT('shifting', fr, shifting_result, PF_OUT);
                            // Stand alone crypto elements search, no prerequisites required
                            case CElement.scheduling do
                                scheduling_result := Find_Scheduling(fr);
                                if scheduling_result['found'] then
                                    Fill_PFOUT('scheduling', fr, scheduling_result, PF_OUT);
                            case CElement.sbox do
                                sbox_result := Find_Sbox(fr);
                                if sbox_result['found'] then
                                    Fill_PFOUT('sbox', fr, sbox_result, PF_OUT);
                            case CElement.mixing do
                                mixing_result := Find_Mixing(fr);
                                if mixing_result['found'] then
                                    Fill_PFOUT('mixing', fr, mixing_result, PF_OUT);
                            otherwise do
                                print("Unsupported Crypto Element")
    return;
```

Sub-Definition 2.1: State
In a given record, the `state` *is the intersection of memory addresses (i.e., memory bases) found in "reads" and "writes" of the same buffer "length", wherein "entropy" is higher or equal on write. The result is known as* `basesIntersection`*. We also collect the bases that only exist in reads and we call them* `uniqueReadBases`*.*

This can be considered the technical definition of an internal state of a crypto algorithm. This is the main concept that we rely on in the task of detecting and identifying most of the crypto elements. We believe it is evident that there must be a state, however it is all about its technical representation. Therefore, a discussion on this is held as part of the limitations of the `Analyzer` module and the "Future work" section below.

Sub-Definition 2.2: Scheduling
In a given record, one of the `uniqueReadBases` *of a scheduling element is the* `key` *to be scheduled or expanded. Meaning, if the "length" of a memory write operation is greater than the length of a read operation–where the address doesn't exist in "writes"– and the entropy on write is higher, then we have a candidate.*

Moreover, we realized that it also satisfies other conditions in most of the cases. For instance, it is the record (or crypto element) that has the least number of `uniqueReadBases`, but we dropped employing this condition in the analysis due to uncertainty. Also, this crypto element displays a very high number of arithmetic operations and loop count. Last but not least, in the cases we observed, the following formula holds true
$writeLength\%maxexecs = uniqueReadLength$, but it may limit `CipherTrace` to certain implementations, so it needs to be carefully experimented with before implementation.

Sub-Definition 2.3: Substitution Box
A record that fulfills the following criteria: it features reading single bytes from memory to substitute a `state` *in-place. A* `state` *is n bytes, and each byte is accessed only once for x number of rounds, wherein x is also the number of records exhibiting the same features.*

In a given record, the memory "reads" should be greater than the "writes", and there should be a `state` base (which may exist in read/write buffers). Moreover, if the aggregate length of the "reads" is equal to the aggregate length of the other `uniqueReadBases`, then that exhibits a multiple-times substitution. We note that "null" bytes has been taken into consideration.

Sub-Definition 2.4: Mixing
A record that fulfills the following criteria: it accesses the same `state` *base as the* `Substitution Box` *and it has the same* `caller`*. The only difference is that, the aggregate length of the "reads" is less than that of the* `uniqueReadBases`*.*

In `Mixing`, we also sort of substitute multiple `state` bytes, but with employing arithmetic operations. This relates to the tandem in `Mixing` and `Shifting` which has been mentioned previously in section 3.2 above.

Sub-Definition 2.5: Shifting
`Mixing` *and* `Shifting` *crypto elements accesses the same memory addresses (i.e., bases) even with the same length, but* `Mixing` *has more "insn_arith" and "llvm_insn_store" statistics compared to* `Shifting`*, due to the associated processing that it carries out.*

`Mixing` and `Shifting` work in-tandem forming a Permutation-step–as mentioned earlier, and in section 3.2 above.

Sub-Definition 2.5: Initial KRound
This is essentially an arithmetic heavy routine, as it performs arithmetic operations on the `state` *and the expanded* `key`*. This occurs for x number of rounds. Therefore, if there is* `Scheduling`*, we expect this crypto element to exist or vice versa.*

In a given record, if the read base is the same as the write base of the `Scheduling` output–wherein buffer length is equal–then we have a candidate. We realized that, this crypto element accesses a `state` base, but not necessarily in the same record being analyzed, but in one of its iterations (i.e., other records of the same element). Also, it may have the same traits as `Substitution`, wherein entropy changes. However, we dropped employing this criteria in the analysis since it needs to be explored further. Last but not least, in the cases we witnessed, the following formula holds true $expandedKeyLength = noOfRounds * keySize$. However, we have refrained from employing this formula in the analysis since it requires more experiments.

Sub-Definition 2.5: KRound
This crypto element depends on the `Initial KRound` *crypto element. In fact, it is almost identical to* `KRound` *except for the former's dependency on the* `Scheduling`*, as it operates directly on the* `state`*, wherein its length must not be 0.*

In closing, the catalyst of successful execution of `FILTERING` and `CRYPTO ELEMENTS IDENTIFICATION` is basically the loop count and the number of arithmetic operations. The enabler for both stages especially the latter is the memory access patterns detected and identified. Not to mention, the main enabler for `CipherTrace` as a whole (and the `Analyzer` module in particular) is the `func_stats` plugin. In the interest of space and for the sake of simplicity, we omit discussing the `REPORTING` stage. The Source of `CipherTrace`, test algorithms and test data are available below.

Memory Re-construction

Generally, this technique is required to perform memory analysis, and specifically in our case, it is required to dump the `key`, `plaintext`, or `ciphertext` by the `Analyzer` module. We refer to them as '`key`', and '`state`' objects respectively. We also utilize memory re-construction in the `Verifier` module in its "default" and "heavy" versions.

In this technique, we reconstruct the memory buffers at different points in the execution and for different memory locations. Basically, we employ it to generate memory dumps at different program counters and for certain function `caller` knowing the address space. These points are known as "tap points"–as described in PANDA's Key concepts above: wherein a tap point is a triplet of caller, program counter, and address space.

In Table 3.2 in section 3.4.1 above, we listed and described the low-level information we collect for each function call as far as memory access patterns are concerned. Therefore, when the `Analyzer` module finds certain crypto elements by analyzing the Synthetic Information, we could easily report the tap point of the crypto element, hence being able to dump the '`key`' and '`state`' objects relative to the crypto element.

In the `Verifier` module, we search in the tap-point's memory buffers for some "searchterms". The buffers are obtained by running the `textfinder` plugin using the tap points reported by the `Analyzer` module. In the `Verifier Heavy` version, instead of searching in each `textfinder` output buffers, we instead perform a bulk `textfinder` run, then we execute the split_taps.py script from PANDA to arrange them, and produce a "dat" file for each tap point's data. So that, we are able to run the "seachterms" on each "dat" file separately to avoid certain semantic issues, bypassing certain memory limitations or discrepancies, and achieving better performance.

4 Evaluation and results

4.1 Intro and overview

In this section, we evaluate the application of the concept, i.e., `CipherTrace`. Therefore, compared to the Related work, we evaluate the ”Instrumentation” stage (i.e., ”Inspection”) from a functionality standpoint. In addition to, the ”Feature Extraction” and the ”Identification” stages. Last but not least, we conclude, reflect, and report the limitations and improvements in the ”Conclusion and reflection” and the ”Future work” sections below. The evaluation is performed in two stages: in the first stage, we evaluate `CipherTrace` against other related tools. And in the second stage, we conduct a pure evaluation of `CipherTrace` itself against the Crypto Algorithms Sample.

4.1.1 CipherTrace vs Other tools

We employ standard algorithms in our evaluation and we list them in the ”Crypto Algorithms Sample” section below. They are all block ciphers, and at least a few of them are not that mainstream (e.g., Serpent), but rather identical (at least conceptually) to a mainstream one (e.g., AES). Serpent was a finalist in the AES contest [4]. So it should be rather easy to detect. Therefore, `serpent256` is the control algorithm in our experimental evaluation. The control algorithm is employed to evaluate `CipherTrace` against other related tools.

Even though SysTaint [61] may be a close implementation to ours, but it is quite far from the idea, concept, application, and purpose. Therefore, we plan to cross-evaluate `CipherTrace` with similar tools that are focused on crypto function (or algorithm) detection in binaries. The evaluation against other tools is mainly to validate the `CipherTrace` prototype. So it is not meant to be a fully-fledged comparison on quality, performance, etc.

4.1.2 CipherTrace vs Algorithms sample

We collect various PANDA-replay traces on two different Operating Systems for both architectures: x86-bit and x64-bit. However we eventually target a specific OS, i.e., Windows. We also collect traces while performing different ”casual” tasks (e.g., running a local program, etc.), and a few more while executing each cryptographic algorithm. This aims to evaluate the resilience of our implementation against noise, i.e., false positives. For example, an arithmetic heavy program that is not executing cryptography. More details on the PANDA-replay traces we recorded can be found in ”Attempt 6 (Evaluation)” section in the Appendix.

4.2 Criteria

Our evaluation criteria is as follows;

Criteria	Description
Scalability	How scalable is the tool?
Limitations	How limiting is the analysis approach or crypto detection technique?
Convenience	How convenient is it to use the tool?
Granularity	What does the tool look for? Key, Implementation, or Cipher
Automation	How many manual steps are required to operate the tool?
Classification	Could it acknowledge or help to specify a cipher class?

Table 4.1: Evaluation criteria

The evaluation scheme is as follows: Low, Medium and High. Low exhibits a low criteria while high exhibits a high criteria. The binary criteria (i.e., Cipher Identification) is a yes/no response. We consider searching for ”a cipher” to be of high-granularity, whereas for ”an implementation” a low-granularity. One of the reasons is that, `K-Hunt` [34] was able to find the `key` without pinpointing the algorithm (or a specific implementation of it), nor the cipher class/type (e.g., block cipher). And that we consider quite low-level and employs a very directed analysis.

4.3 Frameworks and tools

4.3.1 Assessment

Intro

In section 2.1.4 above, we have outlined various related tools, which we delved into their depth in the ”Related work” section. In this section, we evaluate the tools that address the same problem and the ones that are available to use, or at least provide sufficient information and context for a practical evaluation. Therefore, the related work is scoped down to the following:

Subject
kerckhoffs (2010) [24]
aligot (2012) [7]
CryptoHunt (2017) [65]
bacs (2017) [33]
K-Hunt(2018) [34]
SysTaint(2018) [61]

Primarily, `aligot`, `kerckhoffs`, `CryptoHunt`, `bacs` and `K-Hunt` are limited to Intel processors–as mentioned in section 2.1.4 above, and we do not have such a limitation. In our evaluation, we do not take obfuscation into consideration to simplify the evaluation and focus on the key Criteria–besides it is out of scope.

SysTaint

`SysTaint`–compared to `CipherTrace`–has a different purpose, applies a different concept, and adopts different technical mechanisms–as mentioned previously in section 4.1.1 above. Moreover, the published analysis scripts were not sufficient to run a single test to explore its technique in detecting the cryptographic functions–as reported in the Appendix, in ”Attempt 0” section. When it comes to detecting cryptographic algorithms (or even specific

implementations of them), it merely identifies the functions just to taint their output by unique labels in a data collection phase [61], to enable an analyst to look at the Data Flow Graph (DFG) [61]. Moreover, it doesn't differentiate between encryption/decryption, compression, or encoding functions [61]. Therefore, we drop it from our evaluation. Nevertheless, we initiated our evaluation with an experiment to experience its look-and-feel (refer to section A.1.5).

aligot

For a 2MB trace of executing `serpent256`, it performs quite poorly and often runs out-of-memory. Poor performance was still intact even after enhancing some scripts and migrating to 'Python3'. This was still the case even after running Google Compute Engine–for 9 hours–with ultra-high resources. `Kali` [14] authors claim to address these performance issues. However, at the time of writing there was no available `Kali` implementation to evaluate. Therefore, we had to split our `serpent256` trace to be able to run it through the tool. Despite our effort, the tool could not detect it, even knowing that it has `aes128_core` as one of its reference implementations and `serpent256` is quite close. In fact, `serpent256` is almost identical [4]. `aligot` is limited to the algorithms which the tool has reference implementations for [7]. In other words, a different `aes128_core` implementation may not even be detected. Refer to the "Outputs" section below in the Appendix for the execution details.

kerckhoffs

We experienced a few issues with some large traces, however the tool's overall performance was good. `kerckhoffs` reduces the search space by detecting signatures of reference implementations of crypto algorithms. Then it applies heuristics on code level and on a data level. However, its `verifier` is based on reference implementations of crypto algorithms. Additionally, we ran `serpent256` trace through the tool, it identified the implementation as `AES` via constant detection. The tool also suspected that the subject is RC4 (stream cipher), or MD5 (hash algorithm), and both are completely different from what `serpent256` really is, a block cipher [4].

CryptoHunt

`CryptoHunt` has some limitations when it comes to incomplete path coverage. Additionally, it is not well-optimized for performance [65]. It requires a reference loop to be provided as an input to match it against the loops extracted from the trace–since it needs to fuzzy match it [65]. Fundamentally, it matches the extracted boolean formulas with the ones from the reference implementations of the crypto algorithms. In our `aes128` and `serpent256` traces, it produced a dozen loops and we did not have a reference loop to provide–as this is completely against our concept.

bacs

`bacs` is based on reference implementations of certain crypto algorithms. Furthermore, it cannot run on a whole trace, and it requires a decent amount of manual work to operate. In the "Segment Selection" pre-step to the analysis stages, `bacs` relies on splitting the trace to segments [33]. Then, after *manually* selecting a segment [33], it applies some heuristics as part of the signature-based "Feature Extraction" sub-system, which is heavily dependent on constants as features. From which, the output is fed to the "Identification" stage, i.e.,

mode-of-operation identification sub-system. A DFG is utilized in all of its analysis stages. The overall process has to start or end with manual analysis, which is likely in combination with other external tools [33]. Worth to mention that, it whitelists some libraries to be excluded from the ”Instrumentation” stage to decrease the size of the trace and boost the performance [33].

khunt

`khunt` is very key-oriented, its analysis is quite directed towards detecting the insecurely handled `keys`. As it searches for the insecurely handled memory buffers that fulfill certain heuristics, e.g., high arithmetic operations and memory access patterns. It is also not able to report any specific crypto algorithms (e.g,. AES, DES) to which the potential insecure key(s) may belong to [34]. Noteworthy that, we were not able to get hold of the analysis scripts to experiment with.

4.3.2 Benchmark 1: CipherTrace vs Other tools

In the following table, we benchmark the assessed tools against the Criteria defined above in comparison to `CipherTrace`:

Subject	Scalability	Limitations	Convenience	Granularity	Classification
kerckhoffs	M (Issues with some traces)	H (CPU, RefImpl)	M (2 steps required)	Impl	No (Tested)
aligot	L (Very Slow)	H (CPU, RefImpl)	M (2 steps required)	Impl	No (Tested)
CryptoHunt	L (Not optimized for performance)	H (CPU, RefImpl)	L (3 steps required + RefLoop)	Impl	No
bacs	L (Cannot process a full trace)	H (CPU, RefImpl, Manual)	L (∼5 steps required + Manual)	Impl	No
K-Hunt	H (Optimized for performance)	H (CPU, RefImpl)	M (∼3 steps required)	Key	No
CipherTrace	H (Fast approach and technique)	L (Compared to others)	H (∼2 steps required- PANDA)	Cipher	Yes (Tested)

Table 4.2: Tools benchmark.

*Low is denoted as L, Medium is denoted as M, High is denoted as H.

`CipherTrace` is the only tool that could visualize a CFG via its `Visualizer` module. Additionally, it is the only tool that adopts Memory Re-construction in its `Verifier` module, making the verification process more reliable. Last but not least, it is the only one that is able to identify the crypto elements (e.g., S-Box, P-Box, etc) to help derive a classification. By identifying the crypto elements and deriving a classification of a cipher, we prove that our concept is feasible. We used a couple of PANDA plugins in `CipherTrace` as mentioned in section 3.4.1 above. As a result, as far as the dynamic analysis approach is concerned, PANDA provides quite a feature-rich platform which made the Application of our Concept and thesis quite straight-forward. `CipherTrace` is very convenient to use, since operating it has been fully-automated successfully. `CipherTrace` has low design limitations, especially when it comes to the dynamic analysis approach and the technique applied to detect the crypto functions and identify the crypto elements, hence identifying the ciphers. Despite the very rich Synthetic Information that was available to us via our PANDA plugin, we only programmed a few patterns as a kick-off. Additionally, our classification is implicit in its design, driven by the identified crypto elements when grouped together. All in all, there are a few enhancements or improvements that would make `CipherTrace` much better, and they are discussed in the ”Future work” section below. Important to note that, the patterns we programmed do not look for a specific key size or a memory buffer size.

In the following benchmark, we could detect AES in different key sizes. Our `VMI-Heuristics` technique is manifested in the ”Inspection” stage (i.e., `Inspection Engine`). Additionally, the ”Feature Extraction” and ”Identification” stages of analysis are the `Analysis Engine`. We argue that, our concept itself is quite scalable, as it's powered by a high-granularity technique to classify the crypto elements into a cipher. Also, the analysis overall is not ”directed” towards a technically low-level task such as `K-Hunt`[34]. Moreover, ”Scalbility” and ”Cipher Classification” in essence compose the notion of **”Fundamentally Scalable”**. A highly convenient tool with low limitations makes the tool pro-automation by design. We believe that, `CipherTrace` is fundamentally scalable and automated.

4.3.3 Benchmark 2: CipherTrace vs Algorithms sample

In the following table, we are mainly interested in the identification of the crypto elements in our Crypto Algorithms Sample to classify them. The elements are: `Mixing`, `S-Box`, `KRound`, `Scheduling`, `KRoundInit`, and `Shifting`. Therefore, we benchmark `CipherTrace` against the PANDA replays mentioned in ”Attempt 6 (Evaluation)” section in the Appendix–using our plugin with a `stack_limit` of 200:

Subject	Info[1]	Elements ID'd	Buffers Found
AES128 - encrypt.exe - ASID: 30249000 - 1 High-arithmetics caller	Replay size: 803MB func_stats size: 115.3MB Duration: 10 mins mainRecs: 1 mainFunction: 401bb3 AESEncrypt (encrypt.exe) roundRoutine: 401b46 Round (encrypt.exe)	All	Key (auto verified) Plaintext (auto verified) Ciphertext (auto verified)
SERPENT256 - c2.exe - ASID: 31ce0000 - 7 High-arithmetics callers	Replay size: 715.7 MB func_stats size: 112.4 MB Duration: 11-25 mins per caller mainRecs: 1-132 Recs (4/7 are 31-33 Recs) mainFunction: 402094 (main in c2.exe) roundRoutine: 404d58 (printf in c2.exe)	All	Key (auto verified) Plaintext (auto verified*) Ciphertext (auto verified)
TF128 - tf.exe - ASID: 30a98000 - 9 High-arithmetics callers	Replay size: 803.1 MB func_stats size: 133.6 MB Duration: 7-46 mins per caller mainRecs: 1-147 Recs (5/9 are <10 Recs) mainFunction: 401810 (test_sequence in tf.exe)[2] roundRoutine: 40279e (PrepareKey in tf.exe)[3]	No S-BOX	Key (semi-auto verified) Plaintext (semi-auto verified) Ciphertext (n/a)
OPENSSLAES256 - openssl.exe - ASID: 2ebe6000 - 17 High-arithmetics callers	Replay size: 803.3 MB func_stats size: 633.8 MB Duration: 30-761 mins per caller mainRecs: 1-158 Recs (9/17 are 1 Recs) Main Functions (sample) -74f87c60 [4] Round Routines (sample) -74f8bf40 [5]	All	Key (auto verified) Plaintext (n/a) Ciphertext (n/a)
CALCPAINT_NOINTERNET - calc.exe: - ASID: 5b679000 - 3 High-arithmetics callers - mspaint.exe - ASID: 5c4ab000 - 0 High-arithmetics callers	Replay size: 417.7 MB func_stats size: 5.4 GB (Replay utilizes 8gb ram) Duration: 246 mins per caller for "calc" Duration: 427 mins for "mspaint"	None	N/A

Table 4.3: CipherTrace benchmark.

Legend

Auto Verified: The `Verifier` module verified the buffer at the reported tap point.
Auto Verified*: The `Verifier Light` sub-module expected buffers to be found, or the `Verifier Heavy` sub-module was utilized to extensively search for buffers.
Semi-Auto Verified: The output of our plugin assisted in finding the buffers– via disassembly.
n/a: Even PANDA's `stringsearch` plugin did not find any buffers where applicable.

`CipherTrace` identified crypto elements in `aes128`, `serpent256`, `twofish128`, and `opensslaes256`, and helped to classify a cipher. Moreover, it has found and verified all the memory buffers (including the `key`) for `aes128`, `opensslaes256` and `serpent256`. The replay of `opensslaes256` was obtained by recording the use of a widely used

[1]The caller for AES128 is also in SERPENT256, and has one `mainRec`; it is ntdll!RtlRunOnceComplete
[2]This is the actual main function in the program, it runs the full sequence of steps for the algorithm
[3]Due to the pre-computed S-Box, and the design of the algorithm, this is the routine that stands out
[4]Symbols: cryptsp.dll!CryptSetProviderExW+4444
[5]Symbols: cryptsp.dll!CryptSetProviderExW + 21564

software, it was a real life scenario and different from our `aes128` implementation. All AES algorithms we evaluated were fully classified as a block cipher, and their buffers were automatically verified–regardless of implementation or key-size.

`serpent256` which is the control algorithm for the experiment was successfully classified given its identified crypto elements. Also, its buffers were automatically verified despite the different key sizes. One of the crypto elements identified is `S-Box`, denoting Sub-bytes, i.e., the substitution step. `Shifting` and `Mixing` work in-tandem as a permutation step (`P-Box`)–as clarified in section 3.2. Both a `S-Box` and a `P-Box` constitute a Substitution Permutation Network (Product Cipher). Add to that, `CipherTrace` was regarding for a block notion–as explained in section 3.4.2. Therefore, we have successfully **classified** that `serpent256` is a block cipher

In the `twofish128` replay, the `S-Box` crypto element failed to be identified because it is not fixed but pre-computed [49]. This posed some limitations on the results of this replay. The current version of `CipherTrace` does only identify `Fixed S-Box`, and that will be further discussed in the ”Conclusion and reflection” section below.

We have successfully managed to unsheathe some good results with relatively low effort. In the `twofish128` replay, the memory buffers weren't effortlessly found, as it required some manual work. This is due to the current version's limitation of `Pre-computed S-Box`, in addition to the design of the `Randometer` module. Moreover, this particular design in the `Randometer` module is intentional and it has a wide variety of customization options. More details will be further discussed in the ”Conclusion and reflection” section below.

In the decoy replay (`calcpaint_nointernet`), no crypto elements were identified despite the high number of arithmetic operations. That may be considered as a proof-of-resilience against decoys or false-positives–at least to a certain extend.

Moreover, most of the analyses took around 10-30 minutes (for func_stats size ranging from 82mb to 633mb) for a replay with 4GB RAM. The `Verifier` module took around 4 minutes on average. We argue that, this is quite an excellent performance compared to the other tools assessed in section 4.3.1 above. The symbols (i.e., function names) reported for the main functions and round routines were accurate for all the replays, except for the round routine of `serpant` and both of `opensslaes256`–which will be reflected upon below.

4.3.4 Summary

With very good performance, resilience to a decoy replay, and without being fundamentally based on reference implementations of certain crypto algorithms, `CipherTrace` identified encryption functions, crypto routines, and helped to derive a cipher classification after identifying the crypto elements. It correctly helped to classify a block cipher algorithm unlike the other tools. For example in `kerckhoffs`, wherein it failed to identify an algorithm given an almost identical one to the reference implementation used in the tool. `CipherTrace` first identified the crypto elements and that drove the classification of the cipher. Hence, that is the bottom-up approach we took. The bottom-up approach (by design) made crypto operations harder to evade our detection and easier to classify a cipher. That was precisely our main objective as outlined in the section 1.3 previously, and as described in the scenario modeled in section 3 above. Last but not least, that proves the feasibility and scalability of our concept (and `CipherTrace`).

The two-and-only processes of `CipherTrace`: ”Inspection” and ”Analysis” were completely automated. Moreover, `CipherTrace` has lower design limitations, making `CipherTrace` pro-automation by design. The functionality that PANDA offers made `CipherTrace` possible, quite straight-forward, and increased its potential. The `Verifier` module and the `Reporter` sub-module both have quite a high potential. Add to that, applying Memory Re-construction in the `Verifier` module has significantly increased the confidence level in the results. Employing the `Visualizer` module to visualize the function calls and plot a CFG has significantly increased readability of the results to say the least. See Figures A.3,A.4,A.5,A.6 as an example. These figures were generated after exporting the function names from `IDA Freeware`[A.1.5] and providing it as a second input to the `Visualizer` module–along with the graph records to plot.

The `Reporter` sub-module provides quite a verbose functionality, in which it can report all potential `key` ’objects’ and more importantly the `state` ’objects’. In closing, refer to the ”Conclusion and reflection” section below for more information about the limitations of `CipherTrace` and the ”Future work” section below for potential improvements or enhancements.

5 Legal, ethical and social considerations

5.1 Intro

In this section, we discuss the legal, ethical, and social consideration in our research and application. Firstly, we define that laws include statues, regulations, and court decisions [6]. Ethics is concerned with what one should or should not do from a moral or ethical standpoint [6]. Last but not least, a social issue is a problem that influences a group of citizens based on age, demographics, background, or ethnicity, etc. [39]. Social issues may be categorized as valence (the opinion is uni-formally interpret the same way across the society), or positional (the popular opinion is divided across the society).

5.2 This dissertation

The complex of laws, acts, or regulations applicable to our dissertation can be categorized to: privacy or data protection, copyrights or patents infringement, or computer misuse[6]. During our research and `CipherTrace` development, we haven't conducted any personal data collection or analysis, nor did we utilize any third-party data or devices. We also note that, we used our own infrastructure in the research and development. As far as copyrights are concerned, we have cited and stated any third-party research or software that we referred to or used–and we only relied on open-source software. Our samples are synthetic, since we made them up and used publicly available software, wherein they do not contain personal data.

In our dissertation, we have adopted the "ACM Code of Ethics and Professional Conduct" [2], specifically its 2018 revision. We argue that, there are no social implications from our research or `CipherTrace` development in general–unless abused or misused in a targeted manner–which becomes a generic problem of misuse.

In our case, it may have been straight-forward to address the legal and ethical considerations, due to the dependency on open-source software and the employment of a synthetic sample we created. That is unlike the case for [6], which has also shed some light on the challenges facing cybersecurity research.

5.3 Misuse

On one hand, one may unlawfully or unethically record a virtual machine activity and run it through `CipherTrace`. In which case, the adversary will be capable of detecting and identifying crypto operations and highly likely be able to extract the cryptographic `keys`–as per to the defined scope and limitations–to unbare encrypted data, digital rights protection, and communications. Similarly, one may run an arbitrary (and benign) binary program through `CipherTrace` attempting to explore its crypto activities to uncover encrypted data, digital rights protection, and communications. But also, one may run a malware sample

through `CipherTrace`, or a virtual machine trace of a ransomware, which would aid the analysis or even help in neutralizing or recovering from the malware. In essence, and as pretty much in any case, we argue that it is a double-edged sword that was driven by benign motivation and intention (refer to section 1.1), which will aid malware and forensic analysis and help to fight cybercrime. Therefore, we are not responsible for any illegal, unethical, or non-social use of our dissertation or its application (`CipherTrace`).

6 Conclusion and reflection

In this section, we conclude our dissertation and reflect on (and criticize) our concept, dynamic analysis approach, and the crypto detection, identification and classification technique(s). We will also discuss the choice of the platform we used (PANDA).

6.1 Overview

Our main contribution is two-fold: the Concept and thesis and the Application (`CipherTrace`). Our technical contribution is the New PANDA plugin for CipherTrace and the `Analysis Engine`. The `Analysis Engine` provide a unique experience compared to the other related tools. This experience is exhibited in identifying crypto elements, visualization of a CFG, verifying the findings via Memory Re-construction, and offering verbose reporting (if needed) to report more data, e.g., `state` and `key` 'objects'.

Our objective was not to get into an advanced analysis of a CFG, DFG, Parameter Reconstruction, or Loop detection. We depend on PANDA to recognize and employ them to a certain degree, while we focus on applying our concept with good results. The information that allows such an advanced analysis and depth is yet reported by our plugin. We also took constants identification out of scope because that defies the purpose of our concept, as it would result in making our Application more static, fixed, or directed such as a few of the other related tools which were mentioned previously in section 2.2 above.

We categorize the rules of security analysis as environmental attack vectors. For example, an adversary attacking the analysis platform itself. As a result, we dropped the "stealthiness" rule of security analysis. As far as our adversary is concerned, the adversary knows that attacking the platform itself is not a viable option, but detecting it, is possible (i.e., not stealthy).

All in all, hardware virtualization does not guarantee stealthiness. Take `Ether` as an example, wherein its stealthiness has been dis-proven by `Nether`[38]. Generally, virtualization environments could be detected and that could be mitigated, as it is a repeatable and an on-going suspense [20, 42]. To the best of our knowledge, DRAKVUF has managed to achieve stealthiness while attempting to preserve hardware resources via leveraging hardware virtualization extensions [32]. Noteworthy that, the overhead in performance or any noise from instrumentation or introspection might signal to the application that it is being executed in an "analysis" environment. Therefore, it is rather challenging to achieve complete stealthiness. Related to `CipherTrace`, QEMU is what PANDA is based on, and it could also be detected at least in emulation mode [48]. In `CipherTrace`, we were not seeking stealthiness, but mostly isolation and safety–and PANDA fulfills both to the best of our knowledge. And PANDA's functionalities made our Application quite straight-forward–as summarized in section 4.3.4 above.

We employed the `asidstory` plugin from PANDA to identify the address space we wish to examine, and that may be a limitation if the whole replay needs to be analyzed. However, in this case, a linear approach may be taken on the expense of performance, i.e., simply pass all the asids in a given reply to our PANDA plugin as an input.

6.2 Analysis and obfuscation

According to the Automata Theory, as stated by Rice's theorem, computer program analysis is inherently undecidable [27]. This theoretical impossibility is particularly relevant in any adversarial game, specifically in the game of obfuscation. Therefore, obfuscation was out of scope, except for what has been mentioned in the Threat Model.

6.3 Concept, approach and technique

Firstly, the Criteria we selected as a basis for the evaluation aims to highlight our contribution compared to the other related tools from a practical standpoint. This criteria is also very compatible with our Concept and thesis and boosts our Motivation.

Our experimental Evaluation and results have highlighted a high potential for our concept, in which it is fundamentally and technically salable. We also displayed a potential for being resilient to Control Flow obfuscation by design.

Our dynamic analysis approach offered a wide range of possibilities in our analysis. It also made it quite straight-forward. However, it does not fulfill the stealthiness property of security analysis, and `CipherTrace` inherits the platform's Limitations.

In our Application, we heavily relied on the concept of '`state`' to identify many crypto elements. Additionally, in the `Randometer` module, we adopted a single-point-of-entry approach (i.e., by `callers`) to start the analysis from. That is resembled in the Information Measurement tests to identify the `caller` of the candidate crypto functions. Similarly, in the `Analyzer` module, the stack (`recStack`) of the main record(s) we start from (`mainRec`) is considered the only point-of-entry to the `FILTERING` stage of function calls. Despite that we offer highly configurable modules, that introduced some addressable limitations, in which some of them are considered Future work.

6.4 Evaluation and results

In the benchmarks 4.3.2 and 4.3.3, we evaluated `CipherTrace` comparatively with the tools of the Related work, with a sharp focus on the Concept and thesis and applying the Threat Model. `CipherTrace` exceeded our initial expectations, as we managed to unsheathe good results and huge potential with relatively small effort.

- `CipherTrace` has an overall higher performance, it is more convenient to use and it is fully-automated.
- `CipherTrace` is fundamentally scalable (conceptually as well as technically) with some addressable limitations.

- Unlike the comparable Related work, `CipherTrace` is semantics independent, e.g., reference implementation or key-size independent. The `AES` algorithm was classified in two different implementations on the same operating system. The `key` memory buffer was located, extracted, and verified in both implementations regardless of having different key sizes. Also, regardless of the difference in CFG in both implementations, the crypto elements were identified and the ciphers were classified. One implementation was exhibiting a multi-level CFG, and the other adopted a more flattened CFG. The latter was obtained from a widely used library.
- We successfully detected and classified the following block ciphers on Windows: `aes128`, `aes256`, `serpent256` and partially `twofish128`. We were also able to extract and verify the reported memory buffers automatically in all of them except for `twofish128`, regardless of the key-size or the implementation.
- Even on a different operating system (Debian Squeeze), whereas being out of scope, `CipherTrace` produced positive results in despite the semantics gap lesson–refer to ” Attempt 4” section in the Appendix.
- In short, `CipherTrace` can generically classify `Fixed S-Box` block ciphers on Windows, and extract the keys, ciphertexts, and plaintexts.

6.5 Data sample

Our Crypto Algorithms Sample was obtained from various sources to achieve diversity. It was intentionally a mixed bag of algorithms without a specific focus. We do not deny that our sample could be limited in size and could have benefited from a specific focus. Nevertheless, we meant to experimentally evaluate the applicability of Concept and thesis from various angles, so that we could pinpoint its strengths and weaknesses. We adopted a decoy replay in our false-positive test, however we do believe that testing a stream cipher (e.g., RC4) would have contributed a great deal in such a test.

6.6 Limitations and discussion

6.6.1 Our PANDA plugin

- If there are two basic blocks sharing the same number of times they got executed, the plugin will report the first encountered in the ”`maxexecs`” and ”`maxexecs_addr`” fields.
- The function calls are reported by line, therefore to get more results one need to increase the `call_limit` on the expense of storage and performance. We used 200 in our evaluation.
- One one hand, we haven’t tested all the different ways of a function call or all the different ways to call the first basic block of a function. On the other hand, in the various disassembly processes we have went through during development, we realized that we didn’t miss them as they were witnessed in the assembly code we dumped.

6.6.2 Randometer Module

The Randometer module is configured to look for encryption functions then outputs their callers from which we start our analysis. Even though we provide a high degree of customization (via config) in all of our modules, but the prospect of an inappropriate caller that inherently makes the analysis inaccurate, incomplete, or takes longer is not far-fetched. However, benefiting from the fine-grained inspection that PANDA offers, we usually catch the highest level in the stack then delve deeper from there decreasing such potential limitation. PANDA's limitation of fifteen levels of callstack information may make it rewarding for an adversary to let the analysis sink in depth, in which it could only report partial or incomplete results. In our evaluation, we haven't experienced such a limitation–as it may have been lifted.

In the Randometer module we suppressed Shannon's entropy test by default, meaning it will be harder to detect the data buffers that are encrypted and encoded (or compressed) at the same time. However, we address that in the Configurational Future work.

CipherTrace is initially configured to detect and identify an encryption algorithm. In symmetric cryptography, specially in block ciphers, encryption and decryption procedures are inverses [21]. Therefore, as noted in the "Future work" below, a different configuration option (an inverse of the current) provided to the Randometer module will adapt the current version of CipherTrace to look for a decryption function.

6.6.3 Analyzer Module

We learned that, the current version of CipherTrace is only compatible with Fixed S-Box. That is why the S-Box couldn't be identified in the twofish128 replay. In TwoFish algorithm, the 'fill_keyed_sboxes' function fills the the key-dependent S-boxes. The reason is to make the algorithm less vulnerable to reduced-round attacks, because the attacker would not know the S-Box in-advance [49]. In combination with the 'g()' function and how it is split to two variants (key and key-less) for each round per cycle, this broke the pattern that we programmed for a Fixed S-Box.

One of the default criteria we employ to filter the mainRecs is the maximum number of executions of a basic block for a given caller. An adversary may misdirect the analysis by creating a decoy block of code which has an ultra-high loop count. In which case, that is categorized as code obfuscation, which is out of scope and not part of our Threat Model. We only support "maxexecs" or "llvm_bb" fields for the "main records selection" step in the FILTERING stage. Similarly, we adopt memory buffer statistics as one of the feeds to identify certain crypto elements. Therefore, an adversary may flatten the buffers or force the allocation of decoy buffer sizes. However, in the latter we don't expect much influence on the results, as our analysis does not depend on buffer sizes–as proven by Attempt 4 experiment found in the Appendix. Therefore, as far as adopting non-standard key buffers which is a limitation of K-Hunt [34], we may be resilient (by design) to a certain extend to some classes of data obfuscation.

In opensslaes256 replay, the whole implementation was in-line the body of a larger function, and that was a limitation of SysTaint [61]. On one hand, we have successfully identified the crypto elements and classified the cipher and extracted the key. One the other hand, that partially hindered our Reporter module to identify and report the main function and round routine names which comparatively not that important.

We depend on identifying the so-called 'state' memory buffer in the process of identifying some crypto elements. So, if there is no 'state' buffer or if it is obfuscated (which is out of scope), we will fail to identify the crypto element. We could address that by making the 'state' as one of the criteria and not the only one. The current version of CipherTrace is mainly a proof-of-concept. Also, the state affects the Reporter module as far as reporting the plaintext and ciphertext are concerned. Generally, we cannot deny that our **Parameter Re-construction** capability is quite modest, because we simply depend on the memory buffers accessed by the filtered records that fulfill certain Information Measurement criteria. Therefore, this can be further improved to identify tables in memory (i.e., arrays and similar data structures) which would help to address the Pre-computed S-Box in the TwoFish algorithm.

We derive a cipher classification by looking at the filteredStackRecs one-by-one, then we relate each potential record (function call) of an element with another (if required). Therefore, if the records are more divided–as a result of an OS execution strategy–similar to what we experienced in Attempt 4 experiment found in the Appendix, the analysis may become harder. However, that may carry limited consequences since we aggregate the records and group them prior to the CRYPTO ELEMENTS IDENTIFICATION stage of the analysis.

Generally, in the FILTERING stage of the analysis we almost always adopt a '>' not '>=' comparator. Meaning, the first encountered result will be the one returned. For instance, in "main records selection" step for the "maxexecs" or the "llvm_bb" fields. Similarly, in "stack filtering" and "records filtering" steps that follow.

6.6.4 Reporter Module

We identify the main functions and round routines by the frequency of their appearance in the stack, and relative to the identified crypto element's basic block. And since they will need to be applied per round involving a 'state' buffer, we additionally employ the number of executions and the most frequent length of the 'state' buffer. Despite the consequence of such a design and that the Reporter module is not one of our focal points, it produced satisfactory results when it failed, and good results overall. One one hand, it was very close in opensslaes256 replay and in identifying the round routines in serpant256 and tf128 replays. Apart from that, it identified the rest successfully. Moreover, the tap points reported for the plaintext and the ciphertext are affected. That is because reporting tap points depend on identifying the "main" function in addition to the most frequently seen lengths for the 'state' memory buffers.

Generally, reporting the 'state' objects depend on the most frequent lengths of all the identified states, to decrease the number of objects to be reported. The rationale is that, the interesting states are highly likely to be the most accessed ones. It is worth to mention that, this may contribute to inaccurate or incomplete reporting if not set generously–currently it's value is 3, wherein such a design can be improved in future work.

We also report that we have the same limitation explicitly stated in K-Hunt, and also applicable to the other related tools. That is, if data are stored in the CPU registers they won't be found[34]. CipherTrace with its current implementation is unable to process data stored in the CPU registers, i.e., tap points (e.g, the key).

At the time of writing, the current version of CipherTrace only supports finding the key when a Scheduling crypto element is identified–which can be enhanced in future work.

6.6.5 Verifier Module

In addition to the earlier mentioned PANDA Limitations, a discrepancy was detected in the `textfinder` plugin. This discrepancy affects the `Verifier` module and its `Verifier Heavy` sub-module.

The discrepancy is that, the higher the number of input tap points given to the `textfinder` plugin, the more scrambled and dis-ordered the output data buffers will be. This has called for PANDA's split_taps.py script to arrange them to produce a "dat" file for each tap point's data. That is the approach we took in our `Verifier Heavy` sub-module. However, after heavily testing it, we noticed that even though a buffer exists (also expected by our `Verifier Light` sub-module), it cannot be found even after running the `split_taps.py` script.

Depending on the stack level and operating system, the index wherein the data 'byte' could be found in the output of the `textfinder` plugin need to be provided to the `Verifier` module.

We only follow a Memory Re-construction approach in our `Verifier` module, merely as a sanity-check for an experimental version of `CipherTrace`. However, we could adopt a different way in results verification, such as the one followed in [19], in which the potential keys are being tested.

6.7 Miscellany

- We have carried out a basic fool-proof check for the resilience to false-positives. It was mainly a decoy replay and that can be improved.
- False-negatives were not tested. However we believe that `CipherTrace` adopts quite a sensitive analysis, so there is a prospect of a false-positive rather than a false-negative.
- The LLVM-driven statistics produced by the New PANDA plugin for CipherTrace contributed a lot to resolve discrepancies and to verify the identification of crypto elements. Therefore, this is one of the ways to address cross-platform execution such as in Attempt 4 experiment found in the Appendix, in which we observed that the "`llvm_insn_arit`" field was way more accurate than the "`insn_arith`" field.
- Our results are fully-reproducible, thanks to our forensic-like Report in the Appendix.
- We cannot ignore the prospect of bugs in the current version of `CipherTrace`, despite our thorough tests and reviews. However, since the concept has been proven to be sound, feasible, and our results were positive and verifiable, we find that unlikely where it matters.
- The Information Measurement tests that we employed in the `Randometer` module can be improved in the Future work. The initial configuration utilized was simply influenced from [62]. That is because the current version of `CipherTrace` is simply a proof-of-concept.

- In the ”records filtering” step in the `FILTERING` stage, we look for those records which have logged more than one ”`insn_arith`” operation. And that is merely a low-risk kick-off of the algorithm, since we didn't want to risk losing some records by adopting a higher number. At the same time, this particular filter configuration highlights an active record.

6.8 Summary

In our project proposal we promised the following;

- Detect crypto functions and routines following a different dynamic analysis approach and applying different crypto detection and identification technique(s).
- Classify a Product Cipher.
- Determine the depth of information sufficient to apply our concept and making it feasible to find the `key`.
- Report the semantic gaps, granularity gaps, e.g., associated with different platforms, operating systems or implementations.

We argue that we achieved that and more. In our dissertation we followed a bottom-up approach in detecting cryptography and identifying or classifying the utilized cipher in particular via employing an Out-of-VM dynamic analysis approach. Our Motivation highlighted the lack of work in the field in general and the absence of a concept like ours. Equipped with a fundamental understanding of cryptography and particularly block ciphers, we managed to exceed the initial expectations. In our bottom-up approach, we are semantics-independent and we focus on finding the functional crypto elements of block ciphers from an engineering standpoint [21]. We define these constituents as crypto elements. We first identify them, then classify them together to form our understanding of what is going on. Despite that we developed our experimental version of `CipherTrace` mainly with an in-depth understanding of the `aes` algorithm, we managed to classify other block ciphers.

We constructed a Concept and thesis and came up with its Application and its modular design. In `CipherTrace` we provide various configuration options and offer the ability to visualize and verify the results. We also employed Memory Re-construction in the verification of the results. In our Application, we meant to focus on our task, therefore we took advantage of PANDA. Using PANDA made our application very straight-forward and provided the functionality and higher-level semantics that we needed. The heuristics that we defined were applied to identify the crypto elements and derive a cipher classification. That is in addition to a basic CFG analysis to initiate the in-depth analysis. We admit that our decisions along the way may have introduced some limitations that we discussed earlier.

We have a reason to believe that `CipherTrace` could detect `Fixed S-Box` block ciphers on the Windows platform, or at least has a great potential to carry out such a generic task. That was accomplished mainly by employing memory forensics and IOR based on heuristics we defined–after observing a crypto operation–from the Synthetic Information, and in combination with various Information Measurement tests. Last but not least, we note the very high potential exhibited by the our plugin (`func_stats`), as we barely scratched the surface of the information it reports or its depth. In closing, in the Appendix, we present our forensic-like Report to allow for the reproduction of our results–and given the Source of `CipherTrace`.

6.9 Source

`CipherTrace Inspection Engine` (func_stats PANDA plugin 3.4.1) is available as a branch[1]in our fork of PANDA–at the time of writing. The fork resides in our public repository. We have also merged the change[2]with PANDA (master), i.e., the official release. As per to `CipherTrace Analysis Engine`, it is also available in our Github as its own repository[3]; but at the time of writing, that "private" repository adopts an invitation-only access policy. Last but not least, as per to the Crypto Algorithms Sample, they are described in their own section in the Appendix.

[1]https://github.com/mabdelmoez/panda/tree/func_stats
[2]https://github.com/panda-re/panda/pull/801
[3]https://github.com/mabdelmoez/ciphertrace

7 Future work

7.1 Configurational

The design of `CipherTrace` is modular and highly flexible. As we provide a configuration interface so that the user could run `CipherTrace` with different configuration options. There are many combinations that haven't been applied during our experimental evaluation. For example, in the `Randometer` module, we could have given different measures for the Information Measurement tests, wherein we need to 'un-comment' the code responsible for processing the entropy parameters–as we did not need it in our experimental run. Also in the `Analyzer` module, we carry out "stack filtering" step in the `FILTERING` stage utilizing the `functionstack`, in which we could have utilized the `callstack` instead–for a lower-level semantics. Moreover, we could have used different criteria to identify the `mainRecs`, i.e., some different fields, the stack size and/or the entropy measures for the memory buffers. We shall also add the possibility of configuring different (or more) fields to filter by, which is quite important for the "main records selection", "records filtering", and also helps in the "stack filtering" step.

7.2 Technical

We could take more advantage of PANDA callbacks to further automate our "Inspection" process and improve user experience. In other words, we could embed all of our `CipherTrace` modules in another PANDA plugin (`CipherTrace`) that interfaces with our initial PANDA plugin (`func_stats`) via callbacks. And we could utilize the `tapindex` plugin for high-level memory statistics of tap points. Also, we could utilize the `bigrams` plugin for an alternative way to deal with tap points, i.e., clustering.

In section 3.4.2 above, we discussed our Information Measurement tests and where they have been applied. We also mentioned that, in `CipherTrace` we utilize two measurements: Shannon's entropy test and Chi-Square randomness test. Therefore, here we shed some light on the prospect of using different measurements for entropy or randomness, e.g., influenced by [22]. In which, Gan et al. tested comparatively various entropy and randomness tests. Furthermore, they noted that the entropy measures are not sufficient to determine if a series is chaotic, but they could be used to rank multiple series according to the degree of regularity exhibited. They also added, randomness tests provide a likelihood if the input series is randomly distributed rather than the actual values in the test, given the assumption that the null-hypothesis is true (the series is randomly distributed). Therefore, we could either apply different tests to measure information content (entropy) and distribution (randomness), and/or consider them collectively in the `Analyzer` module. That is since we currently carry out only relative comparison with the data at hand, i.e., memory buffers in the same run. We observed how sensitive the Chi-Square test is, similar to the observations that Wang et al. had [62], which makes the Chi-Square test a necessity to be additionally implemented as a

randomness test in our PANDA plugin (`func_stats`), in addition to Shannon's entropy test (the current-and-only test there).

Finally, we could enhance our Classification of crypto algorithms by adding more heuristics. For example, consider the ratio between memory operations and arithmetic operations, or between the LLVM-based and Assembly-based arithmetic operations. As previously mentioned, LLVM-based readings are very information-rich and have quite a high potential. Therefore, we could utilize them more, especially when it comes to addressing architecture-neutral executions. In `SATURN`[23], LLVM optimization has been heavily depended upon to simplify the lifted code in the task of software de-obfuscation and that produced quite some positive results.

7.3 Conceptual

Zhan et al. presented a new approach for VMI which comes with lower overhead [70]. Perhaps if the dynamic analysis platform followed a similar approach, that would decrease the performance overhead in general and the analysis time in particular. Moreover, paralleling execution trace extraction may also be a way to decrease the trace size for offline tracing [64]. In the meanwhile, adaptive program tracing with online CFG support may be a different approach towards decreasing the trace size overall [53].

PANDA leverages whole-system Memory Tainting, which is similar to the other dynamic analysis platforms mentioned in this dissertation. Comparatively, DBI leverages only in-process tainting which is limited to the memory region(s) the process has access to–and that is what the related tools were based on. Bit-level tainting may unleash a lot of capabilities in taint analysis if leveraged by the platform. For example, `TAINTINDUCE` [9] adopts a bit-level approach, which we may argue provides higher accuracy as it observes the state on a bit-level and it is able to deal with direct dependencies. It also offers propagation policy options to decide upon. Meaning, it is a bit more flexible. There are many other interesting taint analysis approaches or concepts, such as `neutaint`[51] which is a neural network powered taint analysis. Additionally, `ProTracer` [37], which its authors claim that it is a better suited concept to deal with Advanced Persistent Threats (ABTs) with low-overhead by alternating between logging and tainting.

In Attempt 0 experiment in the Appendix, before `CipherTrace` takes its current shape and form, we meant to experiment with LLVM and Dynamic Slicing. In which, we feed the output of the `Randometer` module in the form of memory addresses and/or CPU registers to the `Analyzer` module. In order to first dynamically slice the trace, then start the analysis. This avoids going through the "stack filtering" step that we are currently carrying out in our analysis. In other words, we will be running the analysis on lower-level semantics, and that avoids the associated gaps or overhead with the stack filtering approach we follow. By following a Dynamic Slicing approach, we would also have the possibility of tainting the instructions via the `taintedinstr` plugin. Dynamic Slicing has been used in `bacs` [33] to identify the mode of operation, and it was also used in `Saturn` [23] for software obfuscation.

All in all, we have a reason to believe that our concept is applicable to different classes (or modes) of ciphers, as well as to asymmetric cryptography or to cryptosystems in general. That is the result of being a fundamentally scalable concept. Moreover, we believe that the performance of `CipherTrace` could be further improved applying different programming concepts and/or generally improving the current code base.

Bibliography

[1] Accenture. Ninth annual cost of cybercrime study. URL, 2019. Accessed: 2020-10-07.

[2] Ronald E. Anderson. Acm code of ethics and professional conduct. *Commun. ACM*, 35(5):94–99, May 1992.

[3] E. Berrueta, D. Morato, E. Magaña, and M. Izal. A survey on detection techniques for cryptographic ransomware. *IEEE Access*, 7:144925–144944, 2019.

[4] Eli Biham, Ross Anderson, and Lars Knudsen. Serpent: A new block cipher proposal. In Serge Vaudenay, editor, *Fast Software Encryption*, pages 222–238, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[5] Ed Bradford and Lou Mauget. *Linux and Windows Interoperability Guide*. Prentice Hall Professional Technical Reference, 2001.

[6] Aaron Burstein. Conducting cybersecurity research legally and ethically. 01 2008.

[7] Joan Calvet, Jose Fernandez, and Jean-Yves Marion. Aligot: Cryptographic function identification in obfuscated binary programs. pages 169–182, 10 2012.

[8] Li Chen, Salmin Sultana, and Ravi Sahita. Henet: A deep learning approach on intel® processor trace for effective exploit detection. pages 109–115, 05 2018.

[9] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. One engine to serve 'em all: Inferring taint rules without architectural semantics. In *NDSS*, 2019.

[10] Verizon Communications. Data breach investigations report. URL, 2020. Accessed: 2020-10-07.

[11] Keith D. Cooper and Linda Torczon. *Engineering a compiler*. Morgan Kaufmann, 2 edition, 2013.

[12] Coveware. Ransomware costs double in q4 as ryuk, sodinokibi proliferate. URL, 2020. Accessed: 2020-10-07.

[13] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. Decaf++: Elastic whole-system dynamic taint analysis. 09 2019.

[14] Lorenzo De Carli, Ruben Torres, Gaspar Modelo-Howard, Alok Tongaonkar, and Somesh Jha. Kali: Scalable encryption fingerprinting in dynamic malware traces. pages 3–10, 10 2017.

[15] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed). 07 2019.

[16] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, page 289–298, New York, NY, USA, 2013. Association for Computing Machinery.

[17] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, page 51–62, New York, NY, USA, 2008. Association for Computing Machinery.

[18] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, New York, NY, USA, 2015. Association for Computing Machinery.

[19] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan zee (north) bridge: mining memory accesses for introspection. pages 839–850, 11 2013.

[20] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2), March 2008.

[21] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. 01 2010.

[22] Chee Gan and Gerard Learmonth. Comparing entropy with tests for randomness as a measure of complexity in time series. 12 2015.

[23] Peter Garba and Matteo Favaro. Saturn - software deobfuscation framework based on llvm. pages 27–38, 11 2019.

[24] Felix Gröbert. Automatic Identification of Cryptographic Primitives in Software. Diplomarbeit, Ruhr-University Bochum, Germany, 2010.

[25] Andrew Henderson, Aravind Prakash, Lok-Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *ISSTA 2014*, 2014.

[26] Gregory Hill and Xavier Bellekens. Cryptoknight: Generating and modelling compiled cryptographic primitives. *Information (Switzerland)*, 9, 09 2018.

[27] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[28] Diane Duros Hosfelt. Automated detection and classification of cryptographic algorithms in binary programs through machine learning. *ArXiv*, abs/1503.01186, 2015.

[29] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. Diversification and obfuscation techniques for software security: a systematic literature review. *Information and Software Technology*, 07 2018.

[30] Security Intelligence. What's new in the 2019 cost of a data breach report. URL, 2019. Accessed: 2020-10-07.

[31] Julian Kirsch, Zhechko Zhechev, Bruno Bierbaumer, and Thomas Kittel. Pwin – pwning intel pin: Why dbi is unsuitable for security applications. In Javier Lopez, Jianying Zhou, and Miguel Soriano, editors, *Computer Security*, pages 363–382, Cham, 2018. Springer International Publishing.

[32] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[33] Pierre Lestringant. *Identification of cryptographic algorithms in binary programs*. PhD thesis, 12 2017.

[34] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. K-hunt: Pinpointing insecure cryptographic keys from execution traces. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 412–425, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[36] Noe Lutz. Lutz towards revealing attackers ' intent by automatically decrypting network traffic. 2008.

[37] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. 01 2016.

[38] mfaerevaag. mfaerevaag/nether: Detect ether: Malware analysis via hardware virtualization extensions. https://github.com/mfaerevaag/nether, Feb 2020.

[39] C. Wright Mills. *The sociological imagination / C. Wright Mills*. Oxford University Press New York, 1959.

[40] Nicholas Nethercote and Julian Seward. Valgrind a program supervision framework. volume 42, page 89, 06 2007.

[41] A. M. Nguyen, N. Schear, H. Jung, A. Godiyal, S. T. King, and H. D. Nguyen. Mavmm: Lightweight and purpose built vmm for malware analysis. In *2009 Annual Computer Security Applications Conference*, pages 441–450, Dec 2009.

[42] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Comput. Surv.*, 52(5), September 2019.

[43] Jiaye Pan, Yi Zhuang, and Binglin Sun. Bahk: Flexible automated binary analysis method with the assistance of hardware and system kernel. *Security and Communication Networks*, 2020:1–19, 01 2020.

[44] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, Renton, WA, July 2019. USENIX Association.

[45] Om Parkash and Mukesh . Relation between information measures and chi-square statistic. *International Journal of Pure and Apllied Mathematics*, 84, 05 2013.

[46] Grant W. Petty. On Some Shortcomings of Shannon Entropy as a Measure of Information Content in Indirect Measurements of Continuous Variables. *Journal of Atmospheric and Oceanic Technology*, 35(5):1011–1021, 05 2018.

[47] Sergej Proskurin, Tamas Lengyel, Marius Momeu, Claudia Eckert, and Apostolis Zarras. Hiding in the shadows: Empowering arm for stealthy virtual machine introspection. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 407–417, New York, NY, USA, 2018. Association for Computing Machinery.

[48] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In Juan A. Garay, Arjen K. Lenstra, Masahiro Mambo, and René Peralta, editors, *Information Security*, pages 1–18, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[49] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. The twofish encryption algorithm. 09 2000.

[50] Claude E. Shannon and Warren Weaver. A mathematical theory of communication. 1949.

[51] Dongdong She, Yizheng Chen, Baishakhi Ray, and Suman Jana. Neutaint: Efficient dynamic taint analysis with neural networks, 07 2019.

[52] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In R. Sekar and Arun K. Pujari, editors, *Information Systems Security*, pages 1–25, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[53] H. Sun, C. Zhang, H. Li, Z. Wu, L. Wu, and Y. Li. Atos: Adaptive program tracing with online control flow graph support. *IEEE Access*, 7:127495–127510, 2019.

[54] S. Suresh, M. Mohan, C. Thyagarajan, and R. Kedar. Detection of ransomware in emails through anomaly based detection. In D. Jude Hemanth, V. D. Ambeth Kumar, S. Malathi, Oscar Castillo, and Bogdan Patrut, editors, *Emerging Trends in Computing and Expert Technology*, pages 604–613, Cham, 2020. Springer International Publishing.

[55] Symantec. Symantec internet security threat report 2018. URL, 2018. Accessed: 2020-10-07.

[56] Cheng Tan, Xiaoyan Deng, and Lijun Zhang. Identification of block ciphers under cbc mode. *Procedia Computer Science*, 131:65–71, 01 2018.

[57] AV Test. In 2017 every 4.2 seconds a new malware specimen emerges. malware statistics & trends report (2017).sss. URL, 2017. Accessed: 2020-10-07.

[58] AV Test. Malware statistics & trends report (2020). URL, 2020. Accessed: 2020-10-07.

[59] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. Erim: Secure, efficient in-process isolation with protection keys (mpk). 08 2019.

[60] Cybersecurity Ventures. Global cybersecurity spending predicted to exceed $1 trillion from 2017-2021. URL, 06 2019. Accessed: 2020-10-07.

[61] Gabriele Viglianisi, Michele Carminati, Mario Polino, Andrea Continella, and Stefano Zanero. Systaint: Assisting reversing of malicious network communications. pages 1–12, 12 2018.

[62] Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Steal this movie: Automatically bypassing DRM protection in streaming media services. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 687–702, Washington, D.C., 2013. USENIX.

[63] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: Automatic reverse engineering of encrypted messages. In Michael Backes and Peng Ning, editors, *Computer Security – ESORICS 2009*, pages 200–215, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[64] Yuan Xia, Chao Feng, Xing Zhang, Runhao Li, and Chaojing Tang. Parallelization of extracting binary program execution trace offline. pages 184–188, 06 2018.

[65] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. pages 921–937, 05 2017.

[66] Heng Yin and Dawn Song. *Automatic Malware Analysis An Emulator Based Approach*. Springer New York, 2013.

[67] Chang D. Yoo, Yun-Qing Shi, Hyoung Joong. Kim, Alessandro Piva, and Gwangsu Kim. *Digital Forensics and Watermarking: 17th International Workshop, IWDW 2018, Jeju Island, Korea, October 22-24, 2018, Proceedings*. Springer International Publishing, 2019.

[68] A. Young and M. Yung. Cryptovirology: extortion-based security threats and countermeasures. Oakland, CA, USA, USA, 1996. IEEE.

[69] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. Pemu: A pin highly compatible out-of-vm dynamic binary instrumentation framework. volume 50, 06 2014.

[70] D. Zhan, H. Li, L. Ye, H. Zhang, B. Fang, and X. Du. A low-overhead kernel object monitoring approach for virtual machine introspection. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pages 1–6, 2019.

A Appendix

A.1 Report

A.1.1 Lab Environment

The master machine is a Lenovo ThinkPad X1 Yoga (14”, WQHD, Intel Core i7-8550U, 16GB, SSD, 4G), Gen 3. Windows 10 Pro is installed on the machine. An overview of all environments utilized is as follows[1]:

Name	Purpose	OS Edition	Integrity
Win10 Pro x64	Running virtualization environments Evaluation of PIN-based tools Reverse Engineering Disassembly of binaries Testing on ”Bash” some modules of CipherTrace	Windows 10 Pro Version 1909, OS Build 18363.900	Pre-installed OEM
Ubuntu WSL x86_64	Compiling and testing PANDA (our fork’s branch[2], refreshed with upstream) Execution of CipherTrace on a Linux Subsystem	Ubuntu 18.04.2 LTS (Bionic Beaver) Linux 4.4.0-18362-Microsoft x86_64	Microsoft Store [3]
Ubuntu VM x86_64	Compiling and running PANDA (with our changes) Evaluating [65], and executing our CipherTrace	Ubuntu 14.04.6 LTS (Trusty Tahr)[4] Linux 4.4.0-142-generic x86_64	Image Hash [5]
Win7 Pro x86	Evaluation on Windows platform: record the trace using PANDA	X17-59183.iso[6]	ISO Hash [7]
Debian Squeeze x64	Evaluation on Debian (Squeeze) platform: record the trace using PANDA	qcow2 image	QEMU2 images[8]

Table A.1: Lab overview

*MiniGW GNU gdb 7.6.1 on ”Win7 Pro x86” used to debug the executable(s) and resolve symbols.

[1] All software has been obtained from the official websites, otherwise specified
[2] https://github.com/mabdelmoez/panda/tree/func_stats
[3] A Microsoft store App by Canonical Group Limited
[4] https://www.osboxes.org/ubuntu/#ubuntu-14_04-vmware
[5] SHA-256: 3b571f35a42ecda1f93da8235b67fbeb803b399206f2fd89be3c368fd86974d4
[6] https://drive.google.com/uc?export=download&id=0BxJgS33zZl9baG9TQVNjeG9xYnc
[7] SHA-1: d89937df3a9bc2ec1a1486195fd308cd3dade928
[8] https://wiki.qemu.org/Testing/System_Images

The software and tools utilized on the ”Win10 Pro x64” machine are listed below:

Name	Description
DigitalVolcano Hash Tool 1.2	Calculate the checksum of binaries (for checking integrity)
VMware® Workstation 15 Player 15.5.0 build-14665864	Virtualization software for our Ubuntu VM
IDA Freeware Version 7.0.191002 Windows x64	Reverse Engineering and disassembly of binaries
QEMU 2.9.0 [9][10]	An Emulator that performs hardware virtualization, To spin up the environments on our master machine, To prepare them for a PANDA recording.
Intel PIN pin-2.12-58423-msvc10-windows [11] pin-2.14-71313-msvc12-windows	The DBI framework used for the evaluation of PIN-based tools, such as [7, 24] on PIN 2.12, and [33] on PIN 2.14.
Microsoft Visual C++ Express 2010	Compiling PIN tools (e.g., [7, 24]) for evaluation using msvc10
Microsoft Visual Studio Community 2013 with Update 5	Compiling PIN tools (e.g., [33]) for evaluation using msvc12
Visual Studio Code 1.46.0	Multi purpose IDE used for editing our sample crypto algorithms
MinGW/GNUWin32: make-3.81	Used for compiling our sample crypto algorithms on Windows, Using g++ and gcc version 8.2.0 and a utility for MSVC 10.0.
Python 3.8.2	Used for some performance experiments
Git version 2.23.0.windows.1	Version Control
Neo4J Desktop 1.2.4	Graph Database Server to visualize the output of our analysis
glogg v1.1.4-x86_64	Software to browse and search through long or complex log files
Xming 6.9.0.31	X11 display server for Microsoft Windows operating systems. A display output controller for our Ubuntu VM

Table A.2: Win10_Pro_x64 machine

The software and tools utilized on the ”Ubuntu WSL x86_64” environment are listed below:

Our PANDA branch (from fork)[12]built from upstream, after our PR524[13]	gcc & g++ 7.5.0	llvm-3.3
Python 2.7.15+	python pycparser 2.19 & python protobuf 3.13.0	n/a
Ubuntu clang version 3.3-16ubuntu1 (branches/release_33) (based on LLVM 3.3)	protoc 3.0.0	-

Table A.3: Ubuntu WSL x86_64 environment

*The above software should be automatically installed when building PANDA.

The software and tools utilized on the ”Ubuntu VM x86_64” VM are listed below:

PANDA (our dev environment) built at commit with our changes	gcc & g++ 4.8.5	llvm-3.3
Python 2.7.6	python pycparser 2.18 & python protobuf 3.11.2	pin-3.2-81205-gcc-linux
Ubuntu clang version 3.3-16ubuntu1 (branches/release_33) (based on LLVM 3.3)	protoc 2.5.0	-

Table A.4: Ubuntu VM x86_64 environment

*The above software should be automatically installed when building PANDA, except for Intel PIN.

[9]https://qemu.weilnetz.de/w64/2017/qemu-w64-setup-20170420.exe
[10]SHA-256: 30a2bb49c828bfe85bfecce52638c93376b7882ac85e9ece2b5f7162b1c02da1
[11]https://github.com/jingpu/pintools
[12]https://github.com/mabdelmoez/panda/tree/func_stats
[13]https://github.com/panda-re/panda/pull/524

A.1.2 Crypto Algorithms Sample

The sample crypto algorithms which execution was record are as follows:

Name	Changes
AES-128[14]	- Hardcode the key, ”plaintext” and ”ciphertext” in ”encrypt.cpp” and ”decrypt.cpp”. - Add debug messages to manually instrument the binary and print the output in hex.
Twofish-128[15]	- Added a wrapper called ”main.cpp”. - Hardcode the ”key”, ”plaintext” and ”ciphertext”. - Add debug messages to print the output in hex.
Serpent-256[16]	- Change the main function in ”c2.cpp” to hardcode the key, ”plaintext” and ”ciphertext”. - Comment the Command Line Interface (CLI) prompt, and simply just run the program with the hardcoded parameters. - Add debug messages to print the output in hex.
Openssl-AES256	- openssl, 1.1.1d 10th Sep 2019, for Win7 x86. The library[17]was used as it is. Used with the help of a shell script to pass the parameters; ”key”, ”plaintext” and ”ciphertext”. - OpenSSL 0.9.8o 01 Jun 2010, for Debian x64. The library was used as it is from the qcow2 image. Used with the help of a shell script to pass the parameters; ”key”, ”plaintext” and ”ciphertext”.

Table A.5: Crypto algorithms sample

*All the above algorithms are symmetric block ciphers.

A.1.3 Plaintexts, Keys and Ciphertexts

Our main test was to run the message `TESTENC1337` through the encryption routine of `aes128`, `twofish128` and `serpent256`.

The `plaintext` (in hex) is
`54 45 53 54 45 4e 43 31 33 33 37 00 00 00 00 00.`
The `key` (in hex) is `01 04 02 03 01 03 04 0A 09 0B 07 0F 0F 06 03 00.`
The `key` is duplicated one time to match a 32-Bytes key for `serpent256`.
The `ciphertexts` are as follows:

Algorithm	Ciphertext (in hex)
AES 128-Bit	`42 f5 f7 bb 25 c3 9f b9 f0 d0 5e 38 de 1c dd f8`
Twofish 128-Bit	`bd 24 86 16 82 a1 e0 4d 81 6a 9f cc fe b4 4d 25`
Serpent 256-Bit	`64 c2 f1 23 95 98 90 7c 44 19 5e 1f 44 d0 d6 c4`

For our complementary `opensslaes256` test, we ran the message `TESTencWStr1337` through the encryption routine with the secret `secret` producing the following table:

The `plaintext` (in hex)	`54 45 53 54 65 6e 63 57 53 74 72 31 33 33 37`
The `key` (in hex)	`73 65 63 72 65 74 00 00 00 00 00 00 00 00 00 00` `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`
The `ciphertext` (in hex)	`53 61 6c 74 65 64 5f 5f c7 71 d8 99 d2 d7 a0 00` `5d 8d b2 9d 5f 4f 92 9 20 76 7c d2 2f f7 c5`
The `ciphertext` (in base64)	`U2FsdGVkX1/HcdiZ0tegAF2Nsp1fT5IJIHZ80i/3xY=`

[14]https://github.com/ceceww/aes.git
[15]http://www.cartotype.com/downloads/twofish
[16]https://github.com/JasonQSY/serpent.git
[17]https://slproweb.com/download/Win32OpenSSL-1_1_1d.msi

A.1.4 Building and running PANDA

We always trigger a manual build of PANDA in order to have more control over the build process. Generally, we followed the official PANDA guide, additionally the official manual was also quite helpful. After that, we executed the following `alias` commands:

```
$ alias panda2m-system-arm=/home/osboxes/panda-re/panda/build/arm-softmmu/panda-system-arm
$ alias panda2m-system-i386=/home/osboxes/panda-re/panda/build/i386-softmmu/panda-system-i386
$ alias panda2m-system-ppc=/home/osboxes/panda-re/panda/build/ppc-softmmu/panda-system-ppc
$ alias panda2m-system-x86_64=/home/osboxes/panda-re/panda/build/x86_64-softmmu/panda-system-x86_64
```

A.1.5 Attempts

Initially, we have went through many attempts to figure out our best options. This section sheds light on these attempts:

Attempt 0

We did attempt to build SysTaint, which had a few prerequisites such as `protobufc 2.0`, specific `gcc version`, etc. We meant to experience the look-and-feel of the system, however we didn't bother too much since we were missing many analysis scripts to begin with. After we got hold of some of them, we realized that they were not enough to run a single test.

Our initial PANDA builds were very smooth. We built PANDA1 and PANDA2. On PANDA2, we managed to record an `openssl aes-256-cbc` encryption and decryption on Debian (Squeeze) and Windows 7 operating systems. Additionally, we used `stringsearch` plugin to find our '`secret`' in memory. At this point we realized that, running QEMU inside VMWare is quite slow. So, we prepared our VMs using QEMU natively on the Windows 10 machine, then we invoked our test cases while recording from the Ubuntu VM. We also tried the `scissors` plugin to slice the replay, so that running PANDA plugins take less time. However, we didn't use that during our evaluation in section 4. Moreover, we learned that:

- `tainted_instr` plugin uses `taint2` plugin to track taint, as well as `callstack_instr` plugin to provide callstack information whenever tainted branches are encountered.
- `unigrams` plugin uses the `callstack_instr` plugin to group memory accesses into tap points.
- `filetaint` plugin does effectively use `taint` plugin, wherein a mechanism for querying taint on some data in the replay is required. There are some plugins available for this, such as `tainted_instr` and `tainted_branch`.
- `asidstory` plugin depends on the `OSI` plugin to provide Operation System (OS) introspection information.
- `panda_syscalls2` plugin does only support windows_7_x86, windows_xpsp2_x86, windows_xpsp3_x86, linux_x86 and linux_arm.
- `t_stringsearch` plugin applies taint labels to a particular string, whenever it is seen being read from or written to memory using the `stringsearch` plugin.
- `unigram` plugin collects unigram byte statistics (i.e., a histogram of byte values) for each tap point encountered in a replay for memory reads and writes.

We also meant to experiment with an LLVM Trace, as it was quite interesting to explore Dynamic Slicing. In which, we pick a value in the execution trace and ask what instructions were used in computing that value. Unfortunately, it was only available on PANDA1. There was a branch available on PANDA2 but the system build was unstable. Note that, we need PANDA2 due to the dependency on a few plugins–as follows in Attempt 1.

We also managed to experiment a bit with PANDA1, specifically speaking with coverage, textprinter, printstack, fullstack and bigrams plugins. Additionally, we also ported `correlatetaps` and `tapindex` plugins to PANDA2; at this commit and that is PR524. `correlatetaps` plugin groups tap points by adjacency in reads or writes from/to a memory region. Last but not least, the `keyfind` plugin was quite interesting too.

Attempt 1

Starting from this attempt we used our own plugin (`func_stats`) which was influenced by `fn_memlogger` and `fn_composition` plugins from SysTaint. Our plugin is PANDA2 compatible. Essentially, this is **our PANDA base**.

Firstly, we ran the `memorymap` and `tainted_instr` plugins to experiment with them, in addition to the ones in the previous attempt. From that, we realized the following:

- `asidstory` results are structured as: count, pid, name, asid, first, last.
- `coverage` (asid) results are structured as: asid, in kernel, block address, block size.
- `coverage` (process) results are structured as: process name, process id, thread id, in kernel, block address, block size.
- `stringsearch` results are structured as: caller, program counter, asid (CR3 register). Basically, it outputs PANDA tap points.

In this attempt, we used the `stringsearch` plugin to search the memory for the `ciphertext` of `openssl aes-256-cbc` on a Windows replay–as per to section A.1.3 above. Then we extracted the tap point to establish control over the experiment–or as a validity check. We also used PANDA's find_drm.py script along with `unigrams` plugin and with our plugin (`func_stats`). We used the script to find encryption function(s). We observed that, the output of our plugin (functions employing high number of arithmetic operations) matches to an extremely high degree the output of the script (functions that handle high random output but lower random input). That is an encryption function recipe. Manually, and by employing a few plugins such as `textprinter`, we could find a relation between the "`caller`" of the function in our tap point and the "`callers`" from the script's output. We realized that, some contextual information could get a bit blurry as they may have sank into the depth of the callstack. Therefore we learned that, we have to pick the right level of callstack to intervene from.

Attempt 2

Starting from this attempt, we started to experiment with `aes128` as per to section A.1.3 above. We added various debug messages in our plugin and we manually disassembled the binary program (executable) to cross validate results. Moreover, we investigated and corrected the lifecycle of our plugin in the correct order. Our objective was to figure out what happens

for a specific function call. We chose the function that has the highest number of arithmetic operations (reported by our plugin) to start from. The lifecycle has been as follows: basic blocks are being translated until the address of the first basic block of the function which is also the address of the first instruction in that block and the program counter. The `env->panda_guest_pc` is basically the trigger of that block, e.g., 'jnp', 'jmp', 'call', etc. After the block has been translated, the "memory callbacks" are triggered. And if there is an "oncall callback" initialized, it is triggered before the "memory callbacks". Then finally, the block executes and the "memory callbacks" intercept each memory access. After the last block of the function call is executed, the "return callback" is triggered as many times as the number of blocks in the function call.

In the previous attempt, we found a relation between the results of our plugin and the results of the find_drm.py script by function "`caller`". In this attempt, we categorize `callers` to three: a direct 'call', a direct chain call and an indirect chain call. The "`caller`" that matches the script's output is basically a direct chain 'call' which invokes the first basic block of the function, wherein an intermediate 'jne' might be skipped. In the disassembly process, we learned about the other two types of calls. In which, invoking the first block of the function is a direct 'call', whereby an indirect chain call happens after a few 'jmp' instructions.

Attempt 3

In this attempt, we have groomed the information reported by our plugin, as we realized that some information may have been duplicate knowing that:

- The passed program counter (pc) to "memory callbacks", is the same as `cpu->panda_guest_pc` and `get_progpoint`.
- `rr_get_guest_instr_count` is the same as `cpu->rr_guest_instr_count`.
- The first basic block of the function call (i.e., entrypoint) is the same as the entrypoint during the "oncall callback".
- `get_functions` gives function calls (i.e., `functionstack`), whereas `get_callers` gives function callers (i.e., `callstack`). The latter is basically the return address of the whole call.
- Each address in `get_callers` is the start of the block (i.e., `tb->pc`).

We also categorized the synthetic information we report as static or dynamic. On one hand, the static information is the output of the platform-driven or the assembly-driven processing. On the other hand, the dynamic information is the output of the lifted LLVM code, wherein it is quite verbose and has quite a high potential to provide more insights.

We continued to disassemble and match results between our plugin and PANDA's find_drm.py script, basically looking for the `aes128` steps. We took into consideration the memory access patterns (buffers accessed and their size), number of basic blocks per function, loop count and number of arithmetic operations. We identified in our plugin's output some matching memory patterns for the S-Box in AES ('SubBytes') along with some more information. We also correlated that with the script output. We also used `asidstory` and `coverage` plugins to get some high-level information on the address space we are examining and the number of times functions got called and correlate that with our findings.

Attempt 4

In this attempt, we meant to reproduce the previous results but on a different operating system, i.e., Linux (Debian Squeeze). We found that, for the same program some results were affected by the operating system's internals. Due to different stack handling and memory management approaches in different operating systems [5], we faced some issues reporting some memory buffers. Despite such difference, especially in memory allocation (e.g., in number of buffers and their sizes), all the crypto elements were identified on Debian. Also the program CFG was matching to a high degree the one in Windows, especially for the key functions, even if the CFG was a bit scattered on Debian. Refer to Figure A.1 for the CFG of `aes128` on Debian and Figure A.3 for the corresponding one on Windows. We believe that, what we faced was a result of the level of `callstack` (or `functionstack`) which hindered the identification of the correct program counters for the tap points in question, wherein the `key` and `ciphertext` memory buffers are located. This could have been also a side effect of the `Randometer` module's limitations, or that of the `Reporter` module. In the former, an inappropriate `caller` may have been identified, resulting in identifying incorrect program counters for the associated crypto elements' memory buffers. In the latter, the length of the actual buffer was not in the most frequent 3 lengths of '`state`' buffers. Both buffers were later located by employing the output of the New PANDA plugin for CipherTrace. The `plaintext` was 'lightly' verified in the same run via the `Verifier Light` sub-module. The padding of '00' just needed to be removed when searching.

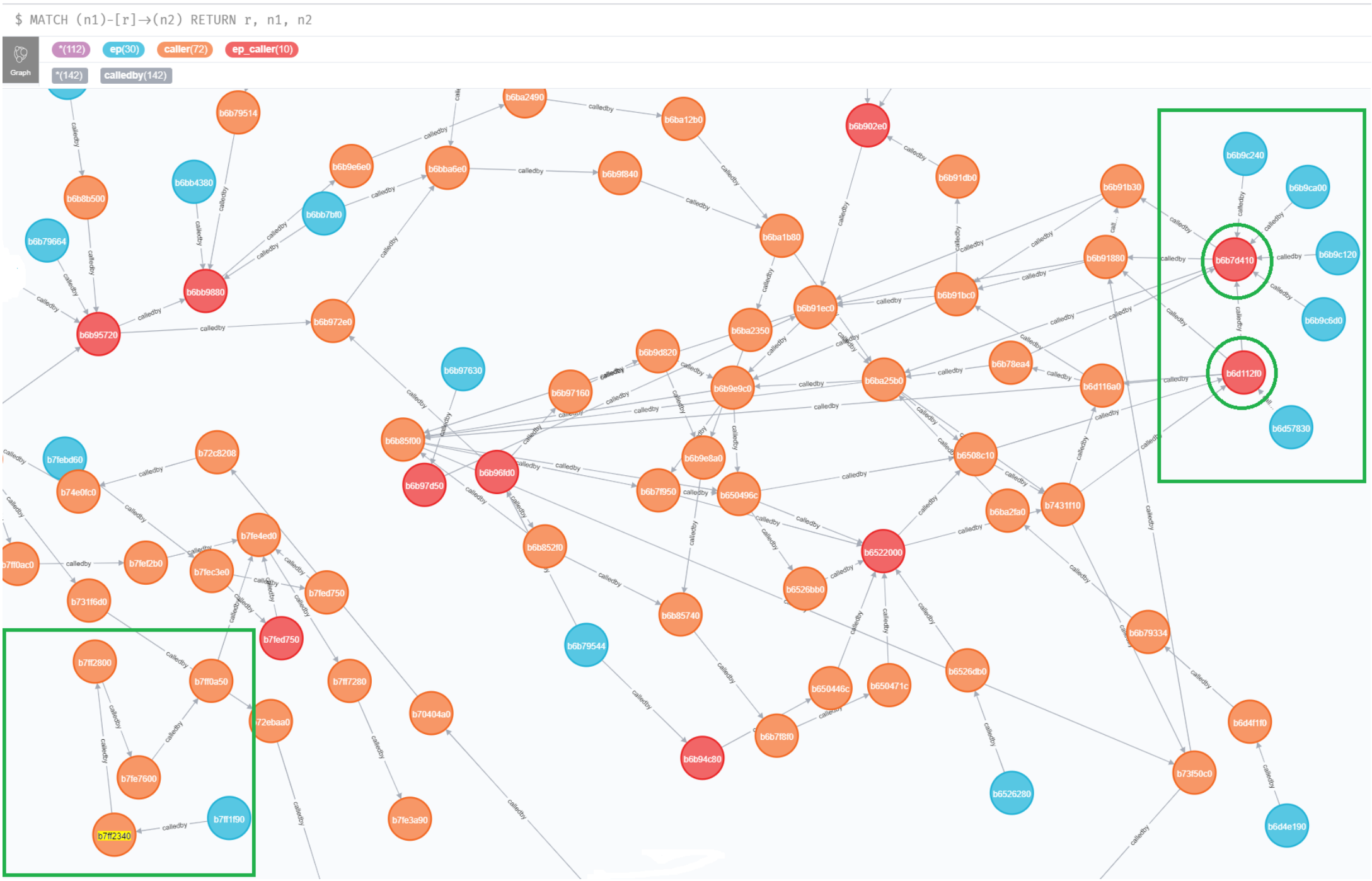

Figure A.1: Debian AES128 CFG

*The biggest shaped ones are the two findings.
*The circled ones are the main finding's main function and round routine.
*The highlighted one is the main function of the second finding.

All in all, such a limitation is quite straight forward to address. We could find better `mainRecs` candidates to start from, and that is a discussion for the Future work. However, we chose to focus on Windows to progress further with our Concept and thesis–as per to our Threat Model.

Attempt 5

In this attempt we managed to come up with some criteria to find different `aes128` steps, and further generalize them to classify a block cipher. Taking into consideration entropy and randomness measures, memory access patterns, number of executions (loop count), and correlation of memory access patterns. Moreover, taking into consideration the sum of the executions of a single basic block and that of a function. All combined has unleashed our Concept and thesis. We also managed to resolve symbols (function names) in a semi-automated manner with the help of `IDA Freeware`. Simply by copying what is displayed in the ”Functions Window” into a so-called `func_db` file. In this attempt, we could even identify the ”KeyExpansion” step and the memory buffers it accesses then dump the `key`.

Attempt 6 (Evaluation)

This attempt marked the start our evaluation. We prepared the following replays in which no connection to the internet was possible unless specified:

1. calcpaint_nointernet: a replay in which microsoft calculator and microsoft paint have been used.
2. aes128: a replay in which aes 128 encryption has been executed.
3. tf128: a replay in which twofish 256 encryption has been executed.
4. serpent256: a replay in which serpent 128 encryption has been executed.
5. opensslaes256: a replay in which `openssl aes-256-cbc` has been executed.

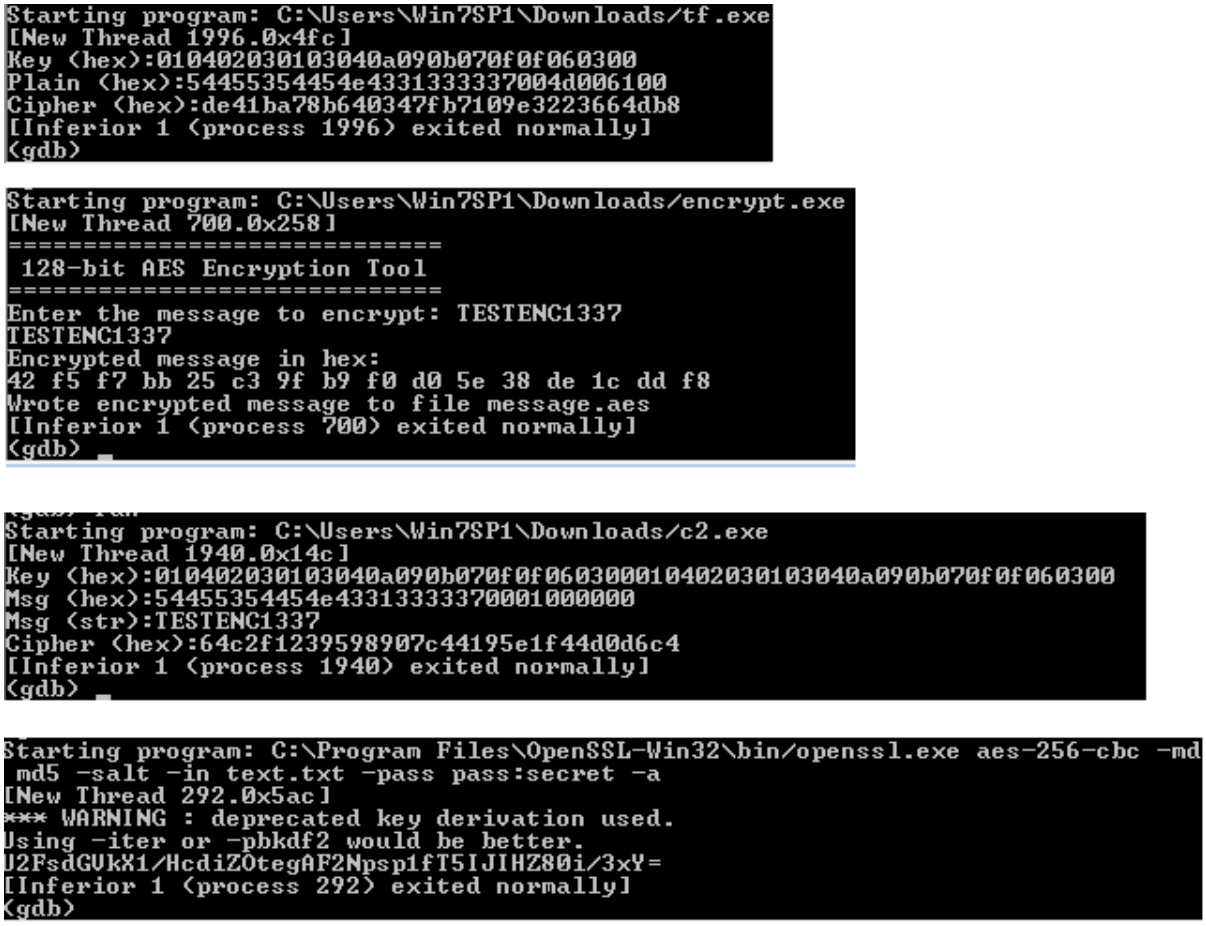

Figure A.2: Execution(s) Screenshot

The tools in scope for evaluation are listed in section 4.3.1 above.

We compiled and built the PIN tools of `aligot` and `kerckhoffs`, both were compiled using `pin-2.12-58423-msvc10-windows` and `VC10` natively on the Windows 10 machine. We simply replaced their code into the ready-to-compile `MyPinTool` which comes with the distribution of Intel PIN–just to avoid any issues with dependencies and 'includes'. The tool is located in the tools directory:
`pin-2.12-58423-msvc10-windows\source\tools`.
As far as the other tools:

- `CryptoHunt`: We directly executed the ready-to-execute tool (32-bit) on the Ubuntu VM using `pin-3.2-81205-gcc-linux`
- `SysTaint`: Refer to Attempt 0 above.
- `k-hunt` and `bacs`: They are addressed as part of the Evaluation and results of this dissertation.

*The PIN_ROOT and PIN_TOOLS environment variables are required to be set.

Refer to section 4 above for more information about this attempt.

A.2 Commands

A.2.1 Linux

1. To extract the tap buffers:

```
$ gunzip -k {tap_buffers}.txt.gz
```

2. To count number of occurrences in a file:

```
$ zgrep -c "{stack}" {tap_buffers}.txt.gz
```

3. To print occurrences in a file:

```
$ grep -E -i -w '{stack}' {randometer_module_output}
```

4. To print occurrences with a format (e.g., high arithmetic in no. of digits):

```
$ awk 'match($0,/[0-9]+ [0-9]{7,} [0-9\.]+/){print $0}' {func_stats_output}
```

5. Replace space by colon in a file (in place):

```
$ sed -i 's/ /:/g' {name}_search_strings.txt
```

6. Set PIN_ROOT Environment Variable on Ubuntu:

```
$ PIN_ROOT=$(pwd)/{pin_dist_location}; export PIN_ROOT
```

7. Compile a cpp program on Ubuntu (32-bit objects):

```
$ g++ -m32 {file}.cpp -o {file}
```

A.2.2 Windows

1. Compile a cpp program on windows (using bash):

```
$ /c/MinGW/bin/g++.exe -g {file}.cpp -o {file}.exe
```

A.2.3 PANDA

Important

1. Run image with no network:

```
$ panda2m-system-i386 -hda win7sp1x86_q2_w86.qcow2 -m 4G -monitor stdio

$ panda2m-system-i386 -hda win7sp1x86_q2_w86.qcow2 -m 4G -monitor stdio -loadvm {
    name}
```

2. Run image with network:

```
$ panda2m-system-i386 -hda win7sp1x86_q2_w86.qcow2 -m 4G -monitor stdio -net nic,
    model=e1000 -net user
```

3. Load image with no network:

4. Run a program via gdb (Go to directory first):

```
$ gdb --args <program e.g., encrypt.exe>
```

5. resolve symbol via gdb (from gdb terminal):

```
$ info symbol 0x{ADDR}
```

6. Begin recording a reply (from QEMU terminal):

```
begin_record {name}
```

7. Run the program (from GDB):

```
run
```

8. End recording a replay (from QEMU terminal):

```
end_record
```

9. Save the snapshot at this state (from QEMU terminal):

```
savevm {name}
```

10. Load a snapshot (from QEMU terminal):

```
loadvm {name}
```

11. Slice a replay:

```
$ panda2m-system-i386 -m 4G -replay {replay_name} -panda scissors:start={
    start_count},end={end_count},name={replay_name_reduced}
```

12. Export assembly code:

```
$ panda2m-system-i386 -m 4G -replay {replay_name} -d in_asm,op,int,rr > asm.out
    2>&1
```

13. Run asidstory:

```
$ panda2m-system-i386 -m 4G -replay {replay_name} -panda osi -os windows-32-7 -
    panda asidstory:width=180
```

14. Run coverage (process):

```
$ panda2m-system-i386 -m 4G -replay {replay_name} -panda osi -os windows-32-7 -
    panda coverage:filename=coverage_proc.csv,mode=process
```

15. Run coverage (asid):

```
$ panda2m-system-i386 -m 4G -replay {replay_name} -panda osi -os windows-32-7 -
    panda coverage:filename=coverage_asid.csv,mode=asid,full=true
```

16. Run unigrams:

```
$ panda2m-system-i386 -m 4G -replay {replay_name} -panda unigrams
```

17. Run func_stats (our plugin):

```
$ panda-system-i386 -m 4G -replay {replay_name} -panda func_stats:asids=0x36766000
    ,hex=true,call_limit=200,stack_limit=16 > func_stats.out
```

Miscellany

1. Run correlatetaps:

```
$ panda2m-system-i386 -m 4G -replay {replay_name} -panda callstack_instr -panda
    correlatetaps
```

2. Run tabindex:

```
$ panda2m-system-i386 -m 4G -replay {replay_name} -panda tapindex
```

   You "read" the output via the script panda-re/panda/panda/scripts/idxmap.py.

3. Run strinsearch:

```
$ panda2m-system-i386 -m 4G -replay {replay_name} -panda callstack_instr -panda
    stringsearch:name={name}
```

   You need to create a file name_search_strings.txt first, that contains your search(es); as string in double quotes, or in hex seperated by colon.

4. Run tainted_instr for a string:

```
$ panda2m-system-i386 -m 4G -replay {replay_name} -panda callstack_instr -panda
    taint2 -panda tainted_instr:summary=false -panda stringsearch:name={name} -
    panda tstringsearch -pandalog output.plog
```

5. Run textprinter for some tapoints:

```
$ panda2m-system-i386 -m 4G -replay {replay_name} -panda callstack_instr -panda
    textprinter
```

   You need to create a file called tap_points.txt first, containing the tap points. With the first line of the file specifying the stack type; 0 (asid), 1 (heuristic), 3 (threaded).

6. Run split_taps.py on textprinter tap buffers (or log file of taps):

```
$ python panda-re/panda/panda/scripts/split_taps.py {tap_buffers}.txt.gz {
    output_prefix}
```

A.2.4 Tools

Aligot

Using PowerShell, from the main directory of the tool:

1. Run the PinTool:

```
$ pin -t C:\dev\pin\pin-2.12-58423-msvc10-windows\source\tools\MyPinTool\Debug\
    MyPinTool.dll -- {executable.exe}
```

2. Run the PinTool (in limited mode):

```
$ pin -t C:\dev\pin\pin-2.12-58423-msvc10-windows\source\tools\MyPinTool\Release\
    MyPinTool.dll -noAPIs -- {executable.exe} -cache 0 -smc_strict 1 -f 2 -
    smc_support 0
```

3. Run the extractor:

```
$ python extraction\main.py {trace.out}
```

   trace.out is the output of running the PinTool. We split it into 4 parts due to its huge size.

kerckhoffs

Using PowerShell, from the 'kerckhoffr' directory of the tool:

1. Run the PinTool:

```
$ pin -t C:\dev\pin\pin-2.12-58423-msvc10-windows\source\tools\MyPinTool-krech\
    Release\krechoffs.dll -- {executable.exe}
```

2. Run the analysis tool (from WSL):

```
$ python __main__.py full {trace.out}
```

   trace.out is the output of running the PinTool.

CryptoHunt

From the main directory of the tool:

1. Run the tracer:

```
$ PIN_ROOT/pin -t tracer/obj-ia32/instracelog.so -- {executable}
```

2. Run loop detection:

```
$ ./loopdetect instrace.txt
```

   instrace.txt is the output of the tracer.

3. Run the detection:

```
$ ./llse {refloop} {targetloop}
```

   You need to provide a reference loop to match with the output from the loop detection step.

A.3 CipherTrace Manual

A.3.1 func_stats PANDA Plugin Input/Output

In the following table, we list all of the input and output of our PANDA plugin (func_stats):

Input	Output
asids: list of address space identifiers to track, separated by '_' endat: instruction count when to end replay, default is 0 call_limit: how many calls to monitor per called function, default is 32 stack_limit: how many enteries to retreive from the callstack, default is 2 hex: print address as hex, default is false (unsigned int 64-bit)	func_stats: a file of records, containing the synthetic info of a function call per line

Table A.6: func_stats PANDA plugin input/output

A.3.2 Analysis Engine Module's Input/Output

In the following table, we list all of the `Analysis Engine` modules' or sub-modules' input and output:

Module	Input	Output
Randometer	**readgram**:unigram reads file **writegram**: unigram writes file **readent**: entropy reads mask **writeent**: entropy writes mask **readrand**: randomness reads mask **writerand**: randomness writes mask **asid**: the ASID to get its caller **ocallers**: output file of callers	**report**: including callers and buffers details **callers**: a file containing a caller per line
Analyzer	**stats**: output file of the func_stats plugin **caller**: the caller to filter by, obtained from the randometer **sym**: symbols file to resolve by (optional) **stack**: stack to filter by (`functionstack` or `callstack`) **conf**: resolve confusion that may result from a verbose report **verbose**: verbose reporting **outprefix**: output files prefix **aconfig**: the config for the analyzer	**report**: including crypto elements found and objects **tappoints**: ".tap" file per caller and type (key or text) **graphs**: ".graph" file per caller
Verifier	**searchterms**: multi line file containing search data (in hex) **readbuffers**: log file containing read tap point data (allows '.gz') **writebuffers**: log file containing write tap point data (allows '.gz') **dataidx**: the data 'byte' index in the buffers array	**report**: including the data series matched
Verifier Light	**tapdir**: directory where ".tap" files reside **matchesfile**: string matches text file	**report**: including the expected matches of data series
Verifier Heavy	**searchterms**: multi line file containing search data (in hex) **datdir**: the directory where ".dat" files reside	**report**: including the data series matched
Visualizer	**graphfile**: the graph file to visualize (output of the analyzer) **sym**: symbols file to resolve by (optional) **conn**: neo4j connection details	**report**: including the graph status and url **graph**: accessed via the url given in the report
Tester	**testdata**: data to test (records, i.e., function calls) **celement**: crypto element name, from the enum in in cryptoelementfinder.py **moduledir**: directoy location of the analyzer module	**report**: including the crypto elements found
Main	**cfgname**: the main config file	runs CipherTrace Analysis Engine (all of the above)

Table A.7: CipherTrace Analyzer input/output

*The "**report**" output refers to the execution report on the standard output, i.e. on the execution environment (e.g., terminal).

A.4 Outputs

A.4.1 Aligot

SERPENT256

Part 1:

```
> Aligot extraction module
> Start: 2020-03-11 11:46:23.848000
---------------------------

> Loop detection... Done
> Garbage collector for invalid loops... Done (26140 loops suppressed)
> Loop I/O... Fail to find the instance!
Loop 9496
We look for 48 at time 5269
Fail to find the instance!
Loop 9496
We look for 48 at time 5269
Done
> Garbage collector for useless loops (no I/O)... Done (1 loops suppressed)
> Loop data flow graph building... WARNING: 1 instruction loop ignored
Done
> Garbage collector for invalid LDFs... Done
> Loop data flow I/O building... Done
> Garbage collector for useless LDFs... Done
> Assigning values to I/O memory parameters... Done
> Dumping results... Done
> Producing graph... ** Limit reached **
Done
---------------------------
```

```
> End: 2020-03-11 11:50:39.687000
```

Part 2:

```
> Aligot extraction module
> Start: 2020-03-11 12:37:48.393000
---------------------------

> Loop detection... Done
> Garbage collector for invalid loops... Done (27412 loops suppressed)
> Loop I/O... Done
> Garbage collector for useless loops (no I/O)... Done (0 loops suppressed)
> Loop data flow graph building... WARNING: 1 instruction loop ignored
WARNING: 1 instruction loop ignored
Done
> Garbage collector for invalid LDFs... Done
> Loop data flow I/O building... Done
> Garbage collector for useless LDFs... Done
> Assigning values to I/O memory parameters... Done
> Dumping results... Done
> Producing graph... ** Limit reached **
Done
---------------------------
> End: 2020-03-11 12:44:25.443000
```

Part 3:

```
> Aligot extraction module
> Start: 2020-03-11 12:37:40.637000
---------------------------

> Loop detection... Done
> Garbage collector for invalid loops... Done (29768 loops suppressed)
> Loop I/O... Done
> Garbage collector for useless loops (no I/O)... Done (0 loops suppressed)
> Loop data flow graph building... WARNING: 1 instruction loop ignored
Done
> Garbage collector for invalid LDFs... Done
> Loop data flow I/O building... Done
> Garbage collector for useless LDFs... Done
> Assigning values to I/O memory parameters... Done
> Dumping results... Done
> Producing graph... ** Limit reached **
Done
---------------------------
> End: 2020-03-11 12:45:13.252000
```

Part 4:

```
> Aligot extraction module
> Start: 2020-03-11 12:37:32.903000
---------------------------

> Loop detection... Done
> Garbage collector for invalid loops... Done (6919 loops suppressed)
> Loop I/O... Done
> Garbage collector for useless loops (no I/O)... Done (0 loops suppressed)
> Loop data flow graph building... WARNING: 1 instruction loop ignored
Done
> Garbage collector for invalid LDFs... Done
> Loop data flow I/O building... Done
> Garbage collector for useless LDFs... Done
> Assigning values to I/O memory parameters... Done
> Dumping results... Done
> Producing graph... ** Limit reached **
Done

---------------------------
> End: 2020-03-11 12:37:56.465000
```

A.4.2 kerckhoffs

TWOFISH128

Failed to parse the trace, and after some fixing, we get `IndexError: list index out of range`

AES128

```
2020-03-24 14:20:38,360 Analysis.py:chains@93 [DEBUG] chainsForImplementation cryptopp_rc4
    0.077535
2020-03-24 14:20:38,361 Analysis.py:chains@93 [DEBUG] chainsForImplementation gladman_aes
    0.168421
2020-03-24 14:20:38,361 Analysis.py:chains@93 [DEBUG] chainsForImplementation openssl_rc4
    0.197183
2020-03-24 14:20:38,362 Analysis.py:chains@93 [DEBUG] chainsForImplementation openssl_aes
    0.145833
2020-03-24 14:20:38,362 Analysis.py:chains@93 [DEBUG] chainsForImplementation openssl_md5
    0.192771
2020-03-24 14:20:38,363 Analysis.py:chains@93 [DEBUG] chainsForImplementation cryptopp_des
    0.071217
2020-03-24 14:20:38,364 Analysis.py:chains@93 [DEBUG] chainsForImplementation cryptopp_md5
    0.085770
2020-03-24 14:20:38,364 Analysis.py:chains@93 [DEBUG] chainsForImplementation cryptopp_rsa
    0.036600
2020-03-24 14:20:38,365 Analysis.py:chains@93 [DEBUG] chainsForImplementation openssl_des
    0.197802
2020-03-24 14:20:38,365 Analysis.py:chains@93 [DEBUG] chainsForImplementation openssl_rsa
    0.072917
2020-03-24 14:20:38,366 Analysis.py:chains@93 [DEBUG] chainsForImplementation beecrypt_aes
    0.220779
2020-03-24 14:20:38,366 Analysis.py:chains@93 [DEBUG] chainsForImplementation cryptopp_aes
    0.069264
2020-03-24 14:20:38,367 Analysis.py:chains@93 [DEBUG] chainsForImplementation beecrypt_md5
    0.170000
2020-03-24 14:20:38,367 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    cryptopp_rc4 0.000000
2020-03-24 14:20:38,368 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    gladman_aes 0.000000
2020-03-24 14:20:38,368 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    openssl_rc4 0.000000
2020-03-24 14:20:38,369 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    openssl_aes 0.000000
2020-03-24 14:20:38,379 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    openssl_md5 0.000000
2020-03-24 14:20:38,380 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    cryptopp_des 0.029412
2020-03-24 14:20:38,381 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    cryptopp_md5 0.000000
2020-03-24 14:20:38,382 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    cryptopp_rsa 0.007114
2020-03-24 14:20:38,382 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    openssl_des 0.000000
2020-03-24 14:20:38,383 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    openssl_rsa 0.012012
2020-03-24 14:20:38,384 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    beecrypt_aes 0.000000
2020-03-24 14:20:38,384 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    cryptopp_aes 0.000000
2020-03-24 14:20:38,385 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    beecrypt_md5 0.000000
2020-03-24 14:20:38,386 Analysis.py:chains@101 [DEBUG] chainsForAlgorithm rc4 0.077068
2020-03-24 14:20:38,386 Analysis.py:chains@101 [DEBUG] chainsForAlgorithm aes 0.061200
2020-03-24 14:20:38,387 Analysis.py:chains@101 [DEBUG] chainsForAlgorithm des 0.070225
2020-03-24 14:20:38,393 Analysis.py:chains@101 [DEBUG] chainsForAlgorithm rsa 0.032795
2020-03-24 14:20:38,401 Analysis.py:chains@101 [DEBUG] chainsForAlgorithm md5 0.077720
2020-03-24 14:20:38,401 Analysis.py:chains@105 [DEBUG] chainsForAlgorithmUnique rc4 0.000000
2020-03-24 14:20:38,402 Analysis.py:chains@105 [DEBUG] chainsForAlgorithmUnique aes 0.000000
2020-03-24 14:20:38,403 Analysis.py:chains@105 [DEBUG] chainsForAlgorithmUnique des 0.019231
2020-03-24 14:20:38,415 Analysis.py:chains@105 [DEBUG] chainsForAlgorithmUnique rsa 0.010409
2020-03-24 14:20:38,417 Analysis.py:chains@105 [DEBUG] chainsForAlgorithmUnique md5 0.000000
2020-03-24 14:20:38,429 Analysis.py:constmemory@109 [DEBUG] a
```

```
2020-03-24 14:20:38,477 Analysis.py:constmemory@111 [DEBUG] b
2020-03-24 14:20:38,497 Analysis.py:constmemory@113 [DEBUG] c
2020-03-24 14:20:38,500 Analysis.py:constmemory@115 [DEBUG] d
2020-03-24 14:20:38,501 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:20:38,501 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:20:38,525 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:20:38,526 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:20:38,536 Analysis.py:constmemory@123 [DEBUG] aes 0.007812 (256)
2020-03-24 14:20:38,548 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:20:38,549 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:20:38,562 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:20:38,584 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:20:38,585 Analysis.py:constmemory@123 [DEBUG] aes 0.007812 (256)
2020-03-24 14:20:38,586 Analysis.py:constmemory@125 [DEBUG] md5 0.333333 (15)
2020-03-24 14:20:38,586 Analysis.py:constmemory@125 [DEBUG] md5 0.000000 (12)
2020-03-24 14:20:38,587 Analysis.py:constmemory@125 [DEBUG] md5 0.000000 (64)
2020-03-24 14:20:38,587 Analysis.py:constmemory@126 [DEBUG] DES_all 0.169231 (65)
2020-03-24 14:20:38,591 Analysis.py:constmemory@127 [DEBUG] RawDES_Spbox 0.049587 (121)
2020-03-24 14:20:38,661 Analysis.py:constmnenomic@141 [DEBUG] unique des 0.111111
2020-03-24 14:20:38,662 Analysis.py:constmnenomic@141 [DEBUG] unique rsa 0.315789
2020-03-24 14:20:38,663 Analysis.py:constmnenomic@141 [DEBUG] unique md5 0.058824
2020-03-24 14:20:38,663 Analysis.py:constmnenomic@146 [DEBUG] intersect des 0.410256
2020-03-24 14:20:38,664 Analysis.py:constmnenomic@146 [DEBUG] intersect rsa 0.392857
2020-03-24 14:20:38,664 Analysis.py:constmnenomic@146 [DEBUG] intersect md5 0.261905
2020-03-24 14:20:38,665 Analysis.py:constmnenomic@152 [DEBUG] implementation rc4 cryptopp
    0.250000
2020-03-24 14:20:38,666 Analysis.py:constmnenomic@152 [DEBUG] implementation rc4 openssl
    0.275000
2020-03-24 14:20:38,666 Analysis.py:constmnenomic@152 [DEBUG] implementation aes beecrypt
    0.217391
2020-03-24 14:20:38,667 Analysis.py:constmnenomic@152 [DEBUG] implementation aes gladman
    0.201439
2020-03-24 14:20:38,668 Analysis.py:constmnenomic@152 [DEBUG] implementation aes cryptopp
    0.242308
2020-03-24 14:20:38,668 Analysis.py:constmnenomic@152 [DEBUG] implementation aes openssl
    0.301370
2020-03-24 14:20:38,669 Analysis.py:constmnenomic@152 [DEBUG] implementation des cryptopp
    0.223938
2020-03-24 14:20:38,677 Analysis.py:constmnenomic@152 [DEBUG] implementation des openssl
    0.174757
2020-03-24 14:20:38,685 Analysis.py:constmnenomic@152 [DEBUG] implementation rsa cryptopp
    0.171806
2020-03-24 14:20:38,688 Analysis.py:constmnenomic@152 [DEBUG] implementation rsa openssl
    0.278107
2020-03-24 14:20:38,700 Analysis.py:constmnenomic@152 [DEBUG] implementation md5 beecrypt
    0.242424
2020-03-24 14:20:38,702 Analysis.py:constmnenomic@152 [DEBUG] implementation md5 cryptopp
    0.237668
2020-03-24 14:20:38,714 Analysis.py:constmnenomic@152 [DEBUG] implementation md5 openssl
    0.191781
[omitted]
2020-03-24 14:20:49,264 Analysis.py:symmetricCipherDataTester@574 [DEBUG] testing a set of 172
     aes keys
2020-03-24 14:20:49,265 Analysis.py:symmetricCipherDataTester@575 [DEBUG] testing a set of 172
     aes inputs
2020-03-24 14:20:49,265 Analysis.py:symmetricCipherDataTester@576 [DEBUG] testing a set of 194
     aes outputs
```

SERPENT256

```
2020-03-24 14:41:20,391 Analysis.py:chains@93 [DEBUG] chainsForImplementation cryptopp_rc4
    0.031809
2020-03-24 14:41:20,392 Analysis.py:chains@93 [DEBUG] chainsForImplementation gladman_aes
    0.073684
2020-03-24 14:41:20,394 Analysis.py:chains@93 [DEBUG] chainsForImplementation openssl_rc4
    0.070423
2020-03-24 14:41:20,395 Analysis.py:chains@93 [DEBUG] chainsForImplementation openssl_aes
    0.055556
2020-03-24 14:41:20,395 Analysis.py:chains@93 [DEBUG] chainsForImplementation openssl_md5
    0.084337
```

```
2020-03-24 14:41:20,396 Analysis.py:chains@93 [DEBUG] chainsForImplementation cryptopp_des
    0.025223
2020-03-24 14:41:20,396 Analysis.py:chains@93 [DEBUG] chainsForImplementation cryptopp_md5
    0.037037
2020-03-24 14:41:20,396 Analysis.py:chains@93 [DEBUG] chainsForImplementation cryptopp_rsa
    0.014758
2020-03-24 14:41:20,397 Analysis.py:chains@93 [DEBUG] chainsForImplementation openssl_des
    0.076923
2020-03-24 14:41:20,397 Analysis.py:chains@93 [DEBUG] chainsForImplementation openssl_rsa
    0.027083
2020-03-24 14:41:20,397 Analysis.py:chains@93 [DEBUG] chainsForImplementation beecrypt_aes
    0.077922
2020-03-24 14:41:20,398 Analysis.py:chains@93 [DEBUG] chainsForImplementation cryptopp_aes
    0.024531
2020-03-24 14:41:20,398 Analysis.py:chains@93 [DEBUG] chainsForImplementation beecrypt_md5
    0.080000
2020-03-24 14:41:20,398 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    cryptopp_rc4 0.000000
2020-03-24 14:41:20,399 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    gladman_aes 0.000000
2020-03-24 14:41:20,399 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    openssl_rc4 0.000000
2020-03-24 14:41:20,399 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    openssl_aes 0.000000
2020-03-24 14:41:20,400 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    openssl_md5 0.000000
2020-03-24 14:41:20,400 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    cryptopp_des 0.000000
2020-03-24 14:41:20,400 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    cryptopp_md5 0.066667
2020-03-24 14:41:20,401 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    cryptopp_rsa 0.006098
2020-03-24 14:41:20,401 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    openssl_des 0.000000
2020-03-24 14:41:20,401 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    openssl_rsa 0.003003
2020-03-24 14:41:20,401 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    beecrypt_aes 0.000000
2020-03-24 14:41:20,401 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    cryptopp_aes 0.000000
2020-03-24 14:41:20,402 Analysis.py:chains@97 [DEBUG] chainsForImplementationUnique
    beecrypt_md5 0.000000
2020-03-24 14:41:20,402 Analysis.py:chains@101 [DEBUG] chainsForAlgorithm rc4 0.031955
2020-03-24 14:41:20,402 Analysis.py:chains@101 [DEBUG] chainsForAlgorithm aes 0.022032
2020-03-24 14:41:20,403 Analysis.py:chains@101 [DEBUG] chainsForAlgorithm des 0.025281
2020-03-24 14:41:20,411 Analysis.py:chains@101 [DEBUG] chainsForAlgorithm rsa 0.012726
2020-03-24 14:41:20,412 Analysis.py:chains@101 [DEBUG] chainsForAlgorithm md5 0.032815
2020-03-24 14:41:20,412 Analysis.py:chains@105 [DEBUG] chainsForAlgorithmUnique rc4 0.000000
2020-03-24 14:41:20,412 Analysis.py:chains@105 [DEBUG] chainsForAlgorithmUnique aes 0.000000
2020-03-24 14:41:20,413 Analysis.py:chains@105 [DEBUG] chainsForAlgorithmUnique des 0.000000
2020-03-24 14:41:20,413 Analysis.py:chains@105 [DEBUG] chainsForAlgorithmUnique rsa 0.005204
2020-03-24 14:41:20,413 Analysis.py:chains@105 [DEBUG] chainsForAlgorithmUnique md5 0.017857
2020-03-24 14:41:20,414 Analysis.py:constmemory@109 [DEBUG] a
2020-03-24 14:41:20,914 Analysis.py:constmemory@111 [DEBUG] b
2020-03-24 14:41:21,147 Analysis.py:constmemory@113 [DEBUG] c
2020-03-24 14:41:21,149 Analysis.py:constmemory@115 [DEBUG] d
2020-03-24 14:41:21,149 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:41:21,150 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:41:21,150 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:41:21,151 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:41:21,151 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:41:21,151 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:41:21,152 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:41:21,152 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:41:21,153 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:41:21,153 Analysis.py:constmemory@123 [DEBUG] aes 0.003906 (256)
2020-03-24 14:41:21,153 Analysis.py:constmemory@125 [DEBUG] md5 0.600000 (15)
2020-03-24 14:41:21,154 Analysis.py:constmemory@125 [DEBUG] md5 0.000000 (12)
2020-03-24 14:41:21,154 Analysis.py:constmemory@125 [DEBUG] md5 0.000000 (64)
2020-03-24 14:41:21,154 Analysis.py:constmemory@126 [DEBUG] DES_all 0.292308 (65)
2020-03-24 14:41:21,155 Analysis.py:constmemory@127 [DEBUG] RawDES_Spbox 0.049587 (121)
2020-03-24 14:41:21,999 Analysis.py:constmnenomic@141 [DEBUG] unique des 0.000000
2020-03-24 14:41:22,000 Analysis.py:constmnenomic@141 [DEBUG] unique rsa 0.210526
```

```
2020-03-24 14:41:22,001 Analysis.py:constmnenomic@141 [DEBUG] unique md5 0.117647
2020-03-24 14:41:22,001 Analysis.py:constmnenomic@146 [DEBUG] intersect des 0.307692
2020-03-24 14:41:22,001 Analysis.py:constmnenomic@146 [DEBUG] intersect rsa 0.285714
2020-03-24 14:41:22,002 Analysis.py:constmnenomic@146 [DEBUG] intersect md5 0.309524
2020-03-24 14:41:22,002 Analysis.py:constmnenomic@152 [DEBUG] implementation rc4 cryptopp
    0.173469
2020-03-24 14:41:22,002 Analysis.py:constmnenomic@152 [DEBUG] implementation rc4 openssl
    0.250000
2020-03-24 14:41:22,003 Analysis.py:constmnenomic@152 [DEBUG] implementation aes beecrypt
    0.152174
2020-03-24 14:41:22,003 Analysis.py:constmnenomic@152 [DEBUG] implementation aes gladman
    0.172662
2020-03-24 14:41:22,003 Analysis.py:constmnenomic@152 [DEBUG] implementation aes cryptopp
    0.138462
2020-03-24 14:41:22,004 Analysis.py:constmnenomic@152 [DEBUG] implementation aes openssl
    0.232877
2020-03-24 14:41:22,004 Analysis.py:constmnenomic@152 [DEBUG] implementation des cryptopp
    0.138996
2020-03-24 14:41:22,004 Analysis.py:constmnenomic@152 [DEBUG] implementation des openssl
    0.155340
2020-03-24 14:41:22,004 Analysis.py:constmnenomic@152 [DEBUG] implementation rsa cryptopp
    0.094714
2020-03-24 14:41:22,004 Analysis.py:constmnenomic@152 [DEBUG] implementation rsa openssl
    0.207101
2020-03-24 14:41:22,005 Analysis.py:constmnenomic@152 [DEBUG] implementation md5 beecrypt
    0.227273
2020-03-24 14:41:22,005 Analysis.py:constmnenomic@152 [DEBUG] implementation md5 cryptopp
    0.161435
2020-03-24 14:41:22,005 Analysis.py:constmnenomic@152 [DEBUG] implementation md5 openssl
    0.205479
[omitted]
2020-03-24 14:46:04,333 Analysis.py:symmetricCipherDataTester@574 [DEBUG] testing a set of 283
     aes keys
2020-03-24 14:46:04,334 Analysis.py:symmetricCipherDataTester@575 [DEBUG] testing a set of 283
     aes inputs
2020-03-24 14:46:04,335 Analysis.py:symmetricCipherDataTester@576 [DEBUG] testing a set of
    2494 aes outputs
```

A.4.3 CipherTrace

AES128

```
[omitted]
Running CipherTrace's Randometer with unigrams output
2020-09-07 17:01:00,043 randometer.py:main@45 [INFO] Starting Randometer for information
    measurement at 2020-09-07 17:01:00.043524
2020-09-07 17:01:00,043 randometer.py:main@46 [INFO] Reading unigram read file
    unigram_mem_read_report.bin
2020-09-07 17:01:01,210 randometer.py:main@52 [INFO] Reading unigram write file
    unigram_mem_write_report.bin
2020-09-07 17:01:01,767 randometer.py:main@60 [INFO] Computing randomness of read buffers
    using Chi-Squared test...
2020-09-07 17:01:01,811 randometer.py:main@63 [INFO] Computing randomness of write buffers
    using Chi-Squared test...
2020-09-07 17:01:01,850 randometer.py:main@68 [INFO] Computing read buffer entropy...
2020-09-07 17:01:01,983 randometer.py:main@71 [INFO] Computing write buffer entropy...
2020-09-07 17:01:02,063 randometer.py:main@74 [INFO] Entropy reads: 20141 writes: 14683
2020-09-07 17:01:02,064 randometer.py:main@77 [INFO] Applying read entropy mask > 0:
2020-09-07 17:01:02,066 randometer.py:main@81 [INFO] Applying write entropy mask > 0:
2020-09-07 17:01:02,068 randometer.py:main@87 [INFO] Applying read rand mask > 10000:
2020-09-07 17:01:02,074 randometer.py:main@91 [INFO] Applying write rand mask < 1000:
2020-09-07 17:01:02,125 randometer.py:main@105 [INFO] Results: reads: 191, writes: 9
[omitted]
2020-09-07 17:01:02,130 randometer.py:main@140 [INFO] Callers for ASID 30249000 are set(['7740
    a5f0'])
2020-09-07 17:01:02,131 randometer.py:main@142 [INFO] Randometer finished measuring
    information of unigram_mem_read_report.bin and unigram_mem_write_report.bin at 2020-09-07
    17:01:02.131108
[omitted]
Running CipherTrace's Analyzer for CALLER 7740a5f0 and REPLAY aes128 and ASID 30249000 with
    func_stats output
```

```
2020-09-07 17:02:11,565 analyzer.py:main@15 [INFO] Starting Analyzer for a 115276032 bytes
    trace func_stats at 2020-09-07 17:02:11.565059
2020-09-07 17:02:11,565 analyzer.py:main@21 [INFO] filtering criteria from config: maxexecs,
    3, 1.0
2020-09-07 17:02:11,566 analyzer.py:main@34 [INFO] Reading the symbols file func_db
2020-09-07 17:02:11,566 analyzer.py:main@40 [INFO] Reading the stats file func_stats
2020-09-07 17:02:16,676 analyzer.py:main@50 [INFO] Filtering by caller
2020-09-07 17:02:16,770 analyzer.py:main@52 [INFO] filteredByCaller count: 3
2020-09-07 17:02:16,770 analyzer.py:main@54 [INFO] Collect the records with fields (maxexecs,
    llvm_bb) that have maximum values
2020-09-07 17:02:16,770 analyzer.py:main@57 [INFO] Find the main stack records (where it all
    begins, with max values of the maxexecs)
2020-09-07 17:02:16,770 analyzer.py:main@60 [INFO] mainRecs count: 1
2020-09-07 17:02:16,770 analyzer.py:main@66 [INFO] Exclude the main records from the filtered
    by caller ones be able to apply stack filtering.
2020-09-07 17:02:16,770 analyzer.py:main@73 [INFO] Do the analysis for each comprehended
    caller stack
2020-09-07 17:02:16,770 analyzer.py:main@97 [INFO] Stack name: functionstack
2020-09-07 17:02:16,771 analyzer.py:main@98 [INFO] Stack filter: [u'828a9420', u'828a90ec', u
    '82897534']
2020-09-07 17:02:16,771 analyzer.py:main@101 [INFO] Filter the records: those which share the
    stack filter, and certain stats (arith>1, loop>1, compc_write_entropy>1.0)
2020-09-07 17:02:16,937 analyzer.py:main@107 [INFO] Create the graph file (for CFG)
2020-09-07 17:02:17,012 analyzer.py:main@116 [INFO] Aggregate by maxexecs and groupby the
    entrypoint (function name)
2020-09-07 17:02:17,013 analyzer.py:main@120 [INFO] Find the crypto elements, as per to
    certain traits
2020-09-07 17:02:17,303 analyzer.py:main@148 [INFO] entrypoint 4014dc
2020-09-07 17:02:17,303 analyzer.py:main@149 [INFO] maxexecs_addr 40152d
2020-09-07 17:02:17,304 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:02:17,304 analyzer.py:main@148 [INFO] entrypoint 4015c1
2020-09-07 17:02:17,304 analyzer.py:main@149 [INFO] maxexecs_addr 4015cf
2020-09-07 17:02:17,304 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:02:17,304 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:02:17,304 analyzer.py:main@171 [INFO] <----------S-Box------------->
2020-09-07 17:02:17,304 analyzer.py:main@183 [INFO] <----------Shifting------------->
2020-09-07 17:02:17,305 analyzer.py:main@148 [INFO] entrypoint 401606
2020-09-07 17:02:17,305 analyzer.py:main@149 [INFO] maxexecs_addr 401613
2020-09-07 17:02:17,305 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:02:17,305 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:02:17,305 analyzer.py:main@171 [INFO] <----------S-Box------------->
2020-09-07 17:02:17,305 analyzer.py:main@183 [INFO] <----------Shifting------------->
2020-09-07 17:02:17,305 analyzer.py:main@148 [INFO] entrypoint 401641
2020-09-07 17:02:17,305 analyzer.py:main@149 [INFO] maxexecs_addr 4016ed
2020-09-07 17:02:17,305 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:02:17,305 analyzer.py:main@183 [INFO] <----------Shifting------------->
2020-09-07 17:02:17,306 analyzer.py:main@148 [INFO] entrypoint 401711
2020-09-07 17:02:17,306 analyzer.py:main@149 [INFO] maxexecs_addr 401b22
2020-09-07 17:02:17,306 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:02:17,306 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:02:17,306 analyzer.py:main@177 [INFO] <----------Mixing------------->
2020-09-07 17:02:17,306 analyzer.py:main@148 [INFO] entrypoint 401bb3
2020-09-07 17:02:17,306 analyzer.py:main@149 [INFO] maxexecs_addr 401bc0
2020-09-07 17:02:17,306 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:02:17,306 analyzer.py:main@148 [INFO] entrypoint 6eb41000
2020-09-07 17:02:17,307 analyzer.py:main@149 [INFO] maxexecs_addr 6eb41028
2020-09-07 17:02:17,307 analyzer.py:main@148 [INFO] entrypoint 6fe41000
2020-09-07 17:02:17,307 analyzer.py:main@149 [INFO] maxexecs_addr 6fe41028
2020-09-07 17:02:17,307 analyzer.py:main@148 [INFO] entrypoint 6fe54ce0
2020-09-07 17:02:17,307 analyzer.py:main@149 [INFO] maxexecs_addr 77309b60
2020-09-07 17:02:17,307 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:02:17,307 analyzer.py:main@148 [INFO] entrypoint 6fe5d514
2020-09-07 17:02:17,307 analyzer.py:main@149 [INFO] maxexecs_addr 6fec3c24
2020-09-07 17:02:17,307 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:02:17,308 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:02:17,308 analyzer.py:main@148 [INFO] entrypoint 6fe7fc9c
2020-09-07 17:02:17,308 analyzer.py:main@149 [INFO] maxexecs_addr 6fe7fdeb
2020-09-07 17:02:17,308 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:02:17,308 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:02:17,308 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:02:17,308 analyzer.py:main@177 [INFO] <----------Mixing------------->
2020-09-07 17:02:17,308 analyzer.py:main@183 [INFO] <----------Shifting------------->
2020-09-07 17:02:17,308 analyzer.py:main@148 [INFO] entrypoint 6fe9e30c
```

```
2020-09-07 17:02:17,308 analyzer.py:main@149 [INFO] maxexecs_addr 6fe9e434
2020-09-07 17:02:17,308 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,308 analyzer.py:main@148 [INFO] entrypoint 6fec4004
2020-09-07 17:02:17,308 analyzer.py:main@149 [INFO] maxexecs_addr 6fec4015
2020-09-07 17:02:17,308 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,308 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 17:02:17,308 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:02:17,308 analyzer.py:main@148 [INFO] entrypoint 6fee9804
2020-09-07 17:02:17,309 analyzer.py:main@149 [INFO] maxexecs_addr 6fee986f
2020-09-07 17:02:17,309 analyzer.py:main@148 [INFO] entrypoint 6fef0a8c
2020-09-07 17:02:17,309 analyzer.py:main@149 [INFO] maxexecs_addr 6fef0b55
2020-09-07 17:02:17,309 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,309 analyzer.py:main@148 [INFO] entrypoint 6fefade4
2020-09-07 17:02:17,309 analyzer.py:main@149 [INFO] maxexecs_addr 6fe5d444
2020-09-07 17:02:17,309 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,309 analyzer.py:main@171 [INFO] <----------S-Box------------>
2020-09-07 17:02:17,309 analyzer.py:main@148 [INFO] entrypoint 763496fb
2020-09-07 17:02:17,309 analyzer.py:main@149 [INFO] maxexecs_addr 76349707
2020-09-07 17:02:17,309 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,309 analyzer.py:main@148 [INFO] entrypoint 7634bb74
2020-09-07 17:02:17,309 analyzer.py:main@149 [INFO] maxexecs_addr 773e4cf3
2020-09-07 17:02:17,309 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,309 analyzer.py:main@148 [INFO] entrypoint 76351292
2020-09-07 17:02:17,309 analyzer.py:main@149 [INFO] maxexecs_addr 76351292
2020-09-07 17:02:17,309 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,310 analyzer.py:main@148 [INFO] entrypoint 7635216e
2020-09-07 17:02:17,310 analyzer.py:main@149 [INFO] maxexecs_addr 7635216e
2020-09-07 17:02:17,310 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,310 analyzer.py:main@148 [INFO] entrypoint 7730a7c3
2020-09-07 17:02:17,310 analyzer.py:main@149 [INFO] maxexecs_addr 7730a824
2020-09-07 17:02:17,310 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,310 analyzer.py:main@148 [INFO] entrypoint 7730c151
2020-09-07 17:02:17,310 analyzer.py:main@149 [INFO] maxexecs_addr 7730c15a
2020-09-07 17:02:17,310 analyzer.py:main@148 [INFO] entrypoint 773132cf
2020-09-07 17:02:17,310 analyzer.py:main@149 [INFO] maxexecs_addr 77313339
2020-09-07 17:02:17,310 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,310 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:02:17,310 analyzer.py:main@148 [INFO] entrypoint 773133a9
2020-09-07 17:02:17,310 analyzer.py:main@149 [INFO] maxexecs_addr 773133f8
2020-09-07 17:02:17,310 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,310 analyzer.py:main@148 [INFO] entrypoint 77313736
2020-09-07 17:02:17,310 analyzer.py:main@149 [INFO] maxexecs_addr 77313755
2020-09-07 17:02:17,311 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,311 analyzer.py:main@148 [INFO] entrypoint 77313869
2020-09-07 17:02:17,311 analyzer.py:main@149 [INFO] maxexecs_addr 773138b4
2020-09-07 17:02:17,311 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,311 analyzer.py:main@148 [INFO] entrypoint 77313f64
2020-09-07 17:02:17,311 analyzer.py:main@149 [INFO] maxexecs_addr 773141e1
2020-09-07 17:02:17,311 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,311 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:02:17,311 analyzer.py:main@148 [INFO] entrypoint 773175e6
2020-09-07 17:02:17,311 analyzer.py:main@149 [INFO] maxexecs_addr 77317652
2020-09-07 17:02:17,311 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,311 analyzer.py:main@148 [INFO] entrypoint 7731c2f9
2020-09-07 17:02:17,311 analyzer.py:main@149 [INFO] maxexecs_addr 7731c2ae
2020-09-07 17:02:17,311 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,311 analyzer.py:main@148 [INFO] entrypoint 77327930
2020-09-07 17:02:17,311 analyzer.py:main@149 [INFO] maxexecs_addr 82878337
2020-09-07 17:02:17,311 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,312 analyzer.py:main@148 [INFO] entrypoint 773cf51a
2020-09-07 17:02:17,312 analyzer.py:main@149 [INFO] maxexecs_addr 773cf555
2020-09-07 17:02:17,312 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,312 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:02:17,312 analyzer.py:main@148 [INFO] entrypoint 773e5b80
2020-09-07 17:02:17,312 analyzer.py:main@149 [INFO] maxexecs_addr 773e5b94
2020-09-07 17:02:17,312 analyzer.py:main@148 [INFO] entrypoint 773e5bd0
2020-09-07 17:02:17,312 analyzer.py:main@149 [INFO] maxexecs_addr 773e5be0
2020-09-07 17:02:17,312 analyzer.py:main@148 [INFO] entrypoint 77402c6a
2020-09-07 17:02:17,312 analyzer.py:main@149 [INFO] maxexecs_addr 77402c6a
2020-09-07 17:02:17,312 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:02:17,312 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:02:17,312 analyzer.py:main@148 [INFO] entrypoint 77402dd6
2020-09-07 17:02:17,312 analyzer.py:main@149 [INFO] maxexecs_addr 77402dd6
```

```
2020-09-07 17:02:17,312 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:02:17,312 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:02:17,312 analyzer.py:main@148 [INFO] entrypoint 774058b5
2020-09-07 17:02:17,312 analyzer.py:main@149 [INFO] maxexecs_addr 774058d4
2020-09-07 17:02:17,313 analyzer.py:main@148 [INFO] entrypoint 77405aec
2020-09-07 17:02:17,313 analyzer.py:main@149 [INFO] maxexecs_addr 773de803
2020-09-07 17:02:17,313 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:02:17,313 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:02:17,313 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:02:17,313 analyzer.py:main@148 [INFO] entrypoint 7740617c
2020-09-07 17:02:17,313 analyzer.py:main@149 [INFO] maxexecs_addr 773cfb9c
2020-09-07 17:02:17,313 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:02:17,313 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:02:17,313 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:02:17,313 analyzer.py:main@148 [INFO] entrypoint 77406f0a
2020-09-07 17:02:17,313 analyzer.py:main@149 [INFO] maxexecs_addr 77406f30
2020-09-07 17:02:17,313 analyzer.py:main@148 [INFO] entrypoint 7740d7fd
2020-09-07 17:02:17,313 analyzer.py:main@149 [INFO] maxexecs_addr 7740d811
2020-09-07 17:02:17,313 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:02:17,313 analyzer.py:main@148 [INFO] entrypoint 7740e3b2
2020-09-07 17:02:17,313 analyzer.py:main@149 [INFO] maxexecs_addr 7740e0d8
2020-09-07 17:02:17,314 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:02:17,314 analyzer.py:main@190 [INFO] Collecting state bases
2020-09-07 17:02:17,314 analyzer.py:main@211 [INFO] Report overall stats
2020-09-07 17:02:17,314 analyzer.py:main@212 [INFO] entrypoint of comprehended caller 76351fb7
2020-09-07 17:02:17,314 analyzer.py:main@223 [INFO] Collect & report routines/functions, max 3
     if there are duplicates
2020-09-07 17:02:17,320 analyzer.py:main@264 [INFO] Overall routine reporting
2020-09-07 17:02:17,321 analyzer.py:main@269 [INFO] Round routine is: Round(uchar ..)
2020-09-07 17:02:17,321 analyzer.py:main@274 [INFO] Main function is: AESEncrypt(uchar ..)
2020-09-07 17:02:17,321 analyzer.py:main@280 [INFO] Collecting tap points data: states (
    ciphertexts or plaintexts)
2020-09-07 17:02:17,322 analyzer.py:main@295 [INFO] Collecting tap points data: key(s)
2020-09-07 17:02:17,322 analyzer.py:main@304 [INFO] Writing tap files (if any)
2020-09-07 17:02:17,323 analyzer.py:main@319 [INFO] Analyzer finished analyzing trace of
    115276032 bytes at 2020-09-07 17:02:17.323287
Duration (excl. verifier): 8 minute(s) for REPLAY aes128 and ASID 30249000 and CALLER 7740a5f0
Running CipherTrace's Verifier Light for REPLAY aes128 and ASID 30249000 with CipherTrace
    Analyzer's output and stringsearch
2020-09-07 17:02:17,869 verifierlight.py:main@9 [INFO] Starting Verifier light at 2020-09-07
    17:02:17.869383
2020-09-07 17:02:17,869 verifierlight.py:main@12 [INFO] Reading matches file
    enc_string_matches.txt
2020-09-07 17:02:17,870 verifierlight.py:main@17 [INFO] Reading .tap files in directory file .
2020-09-07 17:02:17,870 verifierlight.py:main@23 [INFO] Reading tap file ./
    aes128_76351fb7_states.tap
2020-09-07 17:02:17,870 verifierlight.py:main@23 [INFO] Reading tap file ./
    aes128_76351fb7_keys.tap
2020-09-07 17:02:17,870 verifierlight.py:main@36 [DEBUG] Callers found: defaultdict(<type 'int
    '>, {'401f1a': 3, '401f4f': 21})
2020-09-07 17:02:17,871 verifierlight.py:main@37 [DEBUG] PCs found: defaultdict(<type 'int'>,
    {'401500': 1, '401c65': 1, '401bce': 2})
2020-09-07 17:02:17,871 verifierlight.py:main@39 [INFO] The verifier should find data series
2020-09-07 17:02:17,871 verifierlight.py:main@42 [INFO] Verifier light finished searching PCs
    and Callers in 15 matches at 2020-09-07 17:02:17.871426
[omitted]
Running CipherTrace's Verifier with textprinter output
[omitted]
Running textprinter for TAP aes128_76351fb7_keys.tap and REPLAY aes128 and ASID 30249000 with
    CipherTrace Analyzer's output
[omitted]
2020-09-07 17:02:57,965 verifier.py:main@63 [INFO] Data Series 1 was found in the read tap
    buffers, starting from line 15778 till 15794
2020-09-07 17:02:57,972 verifier.py:main@63 [INFO] Data Series 2 was found in the read tap
    buffers, starting from line 15762 till 15778
2020-09-07 17:02:57,991 verifier.py:main@68 [INFO] Verifier finished searching 3 lines in
    buffers at 2020-09-07 17:02:57.991382
Running textprinter for TAP aes128_76351fb7_states.tap and REPLAY aes128 and ASID 30249000
    with CipherTrace Analyzer's output
[omitted]
2020-09-07 17:03:43,039 verifier.py:main@63 [INFO] Data Series 1 was found in the read tap
    buffers, starting from line 0 till 16
2020-09-07 17:03:43,039 verifier.py:main@67 [INFO] Data Series 3 was found in the write tap
```

```
    buffers, starting from line 0 with till 16
2020-09-07 17:03:43,039 verifier.py:main@68 [INFO] Verifier finished searching 3 lines in
    buffers at 2020-09-07 17:03:43.039474
Duration: 10 minute(s) for REPLAY aes128 and ASID 30249000 and Caller 7740a5f0
Duration: 10 minute(s) for REPLAY aes128 and ASID 30249000
```

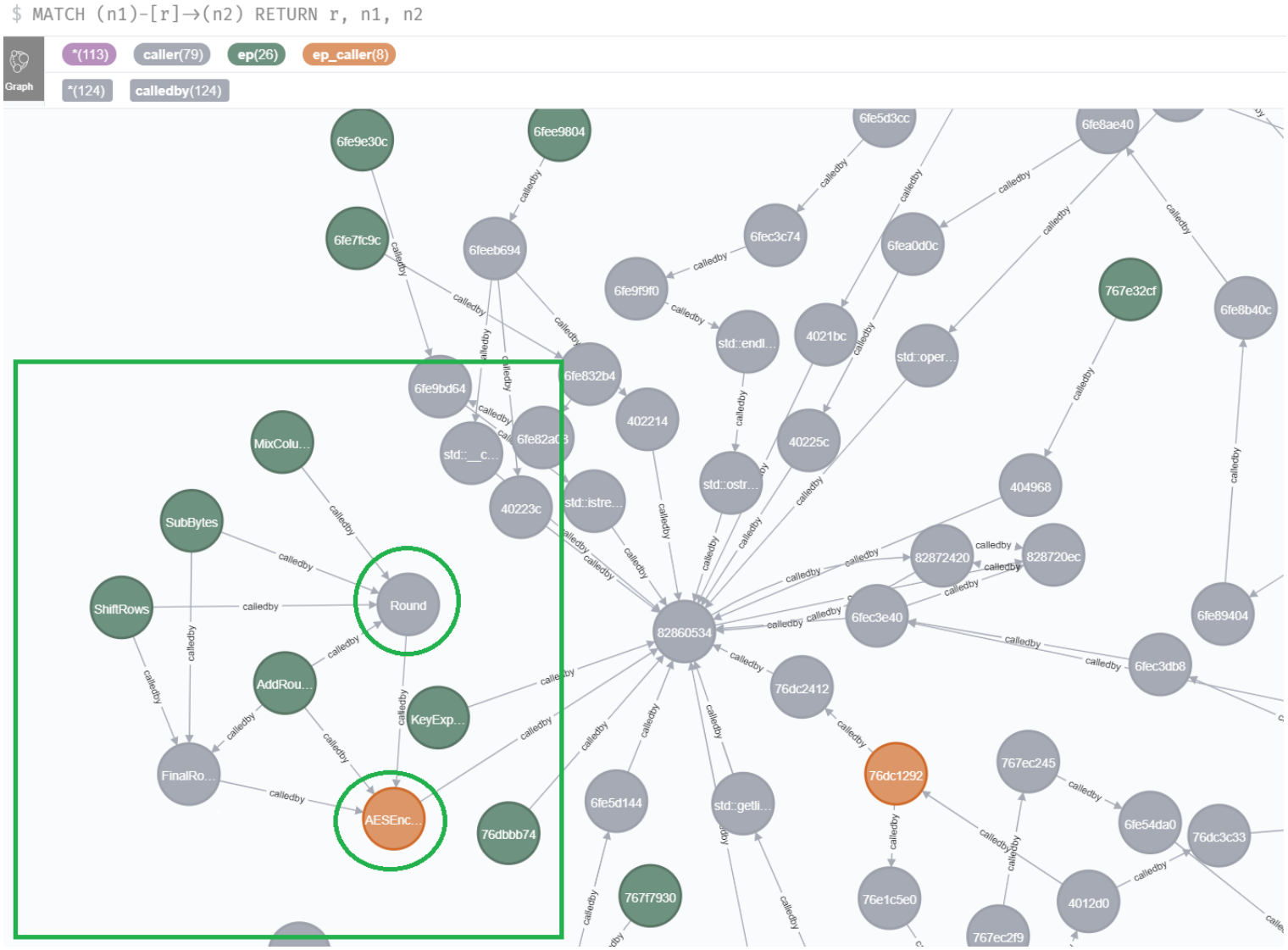

Figure A.3: AES128 CFG

*The biggest shaped one is the finding.
*The circled ones are the finding's main function and round routine.

SERPENT256

```
[omitted]
Running CipherTrace's Randometer with unigrams output
[omitted]
2020-09-07 17:14:54,797 randometer.py:main@45 [INFO] Starting Randometer for information
    measurement at 2020-09-07 17:14:54.797024
2020-09-07 17:14:54,797 randometer.py:main@46 [INFO] Reading unigram read file
    unigram_mem_read_report.bin
2020-09-07 17:14:56,051 randometer.py:main@52 [INFO] Reading unigram write file
    unigram_mem_write_report.bin
2020-09-07 17:14:57,099 randometer.py:main@60 [INFO] Computing randomness of read buffers
    using Chi-Squared test...
2020-09-07 17:14:57,165 randometer.py:main@63 [INFO] Computing randomness of write buffers
    using Chi-Squared test...
2020-09-07 17:14:57,214 randometer.py:main@68 [INFO] Computing read buffer entropy...
2020-09-07 17:14:57,365 randometer.py:main@71 [INFO] Computing write buffer entropy...
2020-09-07 17:14:57,471 randometer.py:main@74 [INFO] Entropy reads: 22499 writes: 16613
2020-09-07 17:14:57,472 randometer.py:main@77 [INFO] Applying read entropy mask > 0:
2020-09-07 17:14:57,475 randometer.py:main@81 [INFO] Applying write entropy mask > 0:
2020-09-07 17:14:57,477 randometer.py:main@87 [INFO] Applying read rand mask > 10000:
2020-09-07 17:14:57,485 randometer.py:main@91 [INFO] Applying write rand mask < 1000:
2020-09-07 17:14:57,554 randometer.py:main@105 [INFO] Results: reads: 360, writes: 37
[omitted]
2020-09-07 17:14:57,584 randometer.py:main@140 [INFO] Callers for ASID 31ce0000 are set(['401
    cde', '401f66', '7740a5f0', '401f80', '4021da', '401c63', '401c97'])
2020-09-07 17:14:57,586 randometer.py:main@142 [INFO] Randometer finished measuring
    information of unigram_mem_read_report.bin and unigram_mem_write_report.bin at 2020-09-07
    17:14:57.586095
Running CipherTrace's Analyzer for CALLER 7740a5f0 and REPLAY serpent128 and ASID 31ce0000
    with func_stats output
2020-09-07 17:17:51,987 analyzer.py:main@15 [INFO] Starting Analyzer for a 112373006 bytes
    trace func_stats at 2020-09-07 17:17:51.987602
2020-09-07 17:17:51,988 analyzer.py:main@21 [INFO] filtering criteria from config: maxexecs,
    3, 1.0
2020-09-07 17:17:51,988 analyzer.py:main@34 [INFO] Reading the symbols file func_db
2020-09-07 17:17:51,988 analyzer.py:main@40 [INFO] Reading the stats file func_stats
2020-09-07 17:17:57,363 analyzer.py:main@50 [INFO] Filtering by caller
2020-09-07 17:17:57,454 analyzer.py:main@52 [INFO] filteredByCaller count: 3
2020-09-07 17:17:57,454 analyzer.py:main@54 [INFO] Collect the records with fields (maxexecs,
    llvm_bb) that have maximum values
2020-09-07 17:17:57,454 analyzer.py:main@57 [INFO] Find the main stack records (where it all
    begins, with max values of the maxexecs)
2020-09-07 17:17:57,455 analyzer.py:main@60 [INFO] mainRecs count: 1
2020-09-07 17:17:57,455 analyzer.py:main@66 [INFO] Exclude the main records from the filtered
    by caller ones be able to apply stack filtering.
2020-09-07 17:17:57,455 analyzer.py:main@73 [INFO] Do the analysis for each comprehended
    caller stack
2020-09-07 17:17:57,455 analyzer.py:main@97 [INFO] Stack name: functionstack
2020-09-07 17:17:57,455 analyzer.py:main@98 [INFO] Stack filter: [u'773f5568', u'773f70b0', u
    '8287b39c']
2020-09-07 17:17:57,455 analyzer.py:main@101 [INFO] Filter the records: those which share the
    stack filter, and certain stats (arith>1, loop>1, compc_write_entropy>1.0)
2020-09-07 17:17:57,631 analyzer.py:main@107 [INFO] Create the graph file (for CFG)
2020-09-07 17:17:57,825 analyzer.py:main@116 [INFO] Aggregate by maxexecs and groupby the
    entrypoint (function name)
2020-09-07 17:17:57,831 analyzer.py:main@120 [INFO] Find the crypto elements, as per to
    certain traits
2020-09-07 17:17:59,304 analyzer.py:main@148 [INFO] entrypoint 401410
2020-09-07 17:17:59,304 analyzer.py:main@149 [INFO] maxexecs_addr 4014ea
2020-09-07 17:17:59,305 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 17:17:59,305 analyzer.py:main@148 [INFO] entrypoint 401569
2020-09-07 17:17:59,305 analyzer.py:main@149 [INFO] maxexecs_addr 40199e
2020-09-07 17:17:59,305 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,305 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:17:59,305 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 17:17:59,305 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:17:59,305 analyzer.py:main@183 [INFO] <----------Shifting------------>
2020-09-07 17:17:59,306 analyzer.py:main@148 [INFO] entrypoint 401a18
2020-09-07 17:17:59,306 analyzer.py:main@149 [INFO] maxexecs_addr 401a26
2020-09-07 17:17:59,306 analyzer.py:main@171 [INFO] <----------S-Box------------>
2020-09-07 17:17:59,306 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:17:59,306 analyzer.py:main@148 [INFO] entrypoint 401c06
2020-09-07 17:17:59,306 analyzer.py:main@149 [INFO] maxexecs_addr 401c3f
2020-09-07 17:17:59,306 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,306 analyzer.py:main@148 [INFO] entrypoint 401da1
2020-09-07 17:17:59,307 analyzer.py:main@149 [INFO] maxexecs_addr 401dae
2020-09-07 17:17:59,307 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 17:17:59,307 analyzer.py:main@148 [INFO] entrypoint 401f3a
2020-09-07 17:17:59,307 analyzer.py:main@149 [INFO] maxexecs_addr 401f47
2020-09-07 17:17:59,307 analyzer.py:main@148 [INFO] entrypoint 402094
```

```
2020-09-07 17:17:59,308 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,308 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 17:17:59,308 analyzer.py:main@148 [INFO] entrypoint 4026c0
2020-09-07 17:17:59,308 analyzer.py:main@149 [INFO] maxexecs_addr 402743
2020-09-07 17:17:59,308 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,308 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:17:59,308 analyzer.py:main@148 [INFO] entrypoint 402bd0
2020-09-07 17:17:59,309 analyzer.py:main@149 [INFO] maxexecs_addr 402b9a
2020-09-07 17:17:59,309 analyzer.py:main@148 [INFO] entrypoint 402f90
2020-09-07 17:17:59,309 analyzer.py:main@149 [INFO] maxexecs_addr 403021
2020-09-07 17:17:59,309 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,309 analyzer.py:main@148 [INFO] entrypoint 403080
2020-09-07 17:17:59,309 analyzer.py:main@149 [INFO] maxexecs_addr 4030ee
2020-09-07 17:17:59,309 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:17:59,309 analyzer.py:main@148 [INFO] entrypoint 4032b0
2020-09-07 17:17:59,309 analyzer.py:main@149 [INFO] maxexecs_addr 82878337
2020-09-07 17:17:59,309 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,309 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:17:59,310 analyzer.py:main@148 [INFO] entrypoint 403660
2020-09-07 17:17:59,310 analyzer.py:main@149 [INFO] maxexecs_addr 403688
2020-09-07 17:17:59,310 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,310 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:17:59,310 analyzer.py:main@148 [INFO] entrypoint 403870
2020-09-07 17:17:59,310 analyzer.py:main@149 [INFO] maxexecs_addr 40388a
2020-09-07 17:17:59,310 analyzer.py:main@148 [INFO] entrypoint 403a00
2020-09-07 17:17:59,310 analyzer.py:main@149 [INFO] maxexecs_addr 403ca5
2020-09-07 17:17:59,310 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,310 analyzer.py:main@148 [INFO] entrypoint 4043c0
2020-09-07 17:17:59,310 analyzer.py:main@149 [INFO] maxexecs_addr 404548
2020-09-07 17:17:59,310 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,310 analyzer.py:main@148 [INFO] entrypoint 404890
2020-09-07 17:17:59,310 analyzer.py:main@149 [INFO] maxexecs_addr 4048c0
2020-09-07 17:17:59,311 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,311 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:17:59,311 analyzer.py:main@148 [INFO] entrypoint 404940
2020-09-07 17:17:59,311 analyzer.py:main@149 [INFO] maxexecs_addr 404983
2020-09-07 17:17:59,311 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,311 analyzer.py:main@148 [INFO] entrypoint 404d38
2020-09-07 17:17:59,311 analyzer.py:main@149 [INFO] maxexecs_addr 77315381
2020-09-07 17:17:59,311 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,311 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:17:59,311 analyzer.py:main@148 [INFO] entrypoint 404d68
2020-09-07 17:17:59,311 analyzer.py:main@149 [INFO] maxexecs_addr 77309b60
2020-09-07 17:17:59,311 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,311 analyzer.py:main@148 [INFO] entrypoint 404e00
2020-09-07 17:17:59,311 analyzer.py:main@149 [INFO] maxexecs_addr 76353d6b
2020-09-07 17:17:59,311 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,312 analyzer.py:main@148 [INFO] entrypoint 404e48
2020-09-07 17:17:59,312 analyzer.py:main@149 [INFO] maxexecs_addr 82878337
2020-09-07 17:17:59,312 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,312 analyzer.py:main@148 [INFO] entrypoint 404e80
2020-09-07 17:17:59,312 analyzer.py:main@149 [INFO] maxexecs_addr 773247c3
2020-09-07 17:17:59,312 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,312 analyzer.py:main@148 [INFO] entrypoint 6eb41000
2020-09-07 17:17:59,312 analyzer.py:main@149 [INFO] maxexecs_addr 6eb41028
2020-09-07 17:17:59,312 analyzer.py:main@148 [INFO] entrypoint 6eb41260
2020-09-07 17:17:59,312 analyzer.py:main@149 [INFO] maxexecs_addr 6eb41278
2020-09-07 17:17:59,312 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,312 analyzer.py:main@148 [INFO] entrypoint 6fe41000
2020-09-07 17:17:59,312 analyzer.py:main@149 [INFO] maxexecs_addr 6fe41028
2020-09-07 17:17:59,312 analyzer.py:main@148 [INFO] entrypoint 6fe54ce0
2020-09-07 17:17:59,312 analyzer.py:main@149 [INFO] maxexecs_addr 7730aba6
2020-09-07 17:17:59,313 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,313 analyzer.py:main@148 [INFO] entrypoint 6fe54ce8
2020-09-07 17:17:59,313 analyzer.py:main@149 [INFO] maxexecs_addr 773179b0
2020-09-07 17:17:59,313 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,313 analyzer.py:main@148 [INFO] entrypoint 6fe68f5c
2020-09-07 17:17:59,313 analyzer.py:main@149 [INFO] maxexecs_addr 82878337
2020-09-07 17:17:59,313 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,313 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 17:17:59,313 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:17:59,313 analyzer.py:main@148 [INFO] entrypoint 6fe6dda0
2020-09-07 17:17:59,313 analyzer.py:main@149 [INFO] maxexecs_addr 6fe6ddb7
```

```
2020-09-07 17:17:59,313 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,313 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:17:59,313 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:17:59,313 analyzer.py:main@148 [INFO] entrypoint 6fea0244
2020-09-07 17:17:59,314 analyzer.py:main@149 [INFO] maxexecs_addr 82878337
2020-09-07 17:17:59,314 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:17:59,314 analyzer.py:main@148 [INFO] entrypoint 6fed7fd8
2020-09-07 17:17:59,314 analyzer.py:main@149 [INFO] maxexecs_addr 82878337
2020-09-07 17:17:59,314 analyzer.py:main@148 [INFO] entrypoint 6fee9804
2020-09-07 17:17:59,314 analyzer.py:main@149 [INFO] maxexecs_addr 6fee986f
2020-09-07 17:17:59,314 analyzer.py:main@148 [INFO] entrypoint 6fef9fd0
2020-09-07 17:17:59,314 analyzer.py:main@149 [INFO] maxexecs_addr 82878337
2020-09-07 17:17:59,314 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,314 analyzer.py:main@148 [INFO] entrypoint 75766788
2020-09-07 17:17:59,314 analyzer.py:main@149 [INFO] maxexecs_addr 773e4cf3
2020-09-07 17:17:59,314 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,314 analyzer.py:main@148 [INFO] entrypoint 757680b0
2020-09-07 17:17:59,314 analyzer.py:main@149 [INFO] maxexecs_addr 757681bc
2020-09-07 17:17:59,315 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,315 analyzer.py:main@148 [INFO] entrypoint 7576c54e
2020-09-07 17:17:59,315 analyzer.py:main@149 [INFO] maxexecs_addr 7576c5b7
2020-09-07 17:17:59,315 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,315 analyzer.py:main@148 [INFO] entrypoint 7578039c
2020-09-07 17:17:59,315 analyzer.py:main@149 [INFO] maxexecs_addr 75780455
2020-09-07 17:17:59,315 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,318 analyzer.py:main@148 [INFO] entrypoint 7730a7c3
2020-09-07 17:17:59,319 analyzer.py:main@149 [INFO] maxexecs_addr 7730a824
2020-09-07 17:17:59,319 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,319 analyzer.py:main@148 [INFO] entrypoint 7730ac15
2020-09-07 17:17:59,319 analyzer.py:main@149 [INFO] maxexecs_addr 7730ac15
2020-09-07 17:17:59,319 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,319 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:17:59,319 analyzer.py:main@148 [INFO] entrypoint 7730ae77
2020-09-07 17:17:59,319 analyzer.py:main@149 [INFO] maxexecs_addr 7730aed6
2020-09-07 17:17:59,319 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,319 analyzer.py:main@148 [INFO] entrypoint 7730c151
2020-09-07 17:17:59,319 analyzer.py:main@149 [INFO] maxexecs_addr 7730c15a
2020-09-07 17:17:59,319 analyzer.py:main@148 [INFO] entrypoint 7730ccfd
2020-09-07 17:17:59,319 analyzer.py:main@149 [INFO] maxexecs_addr 7730ce4e
2020-09-07 17:17:59,320 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,320 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:17:59,320 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:17:59,320 analyzer.py:main@177 [INFO] <----------Mixing------------->
2020-09-07 17:17:59,320 analyzer.py:main@183 [INFO] <----------Shifting------------->
2020-09-07 17:17:59,320 analyzer.py:main@148 [INFO] entrypoint 7730f574
2020-09-07 17:17:59,320 analyzer.py:main@149 [INFO] maxexecs_addr 7730f5a1
2020-09-07 17:17:59,320 analyzer.py:main@148 [INFO] entrypoint 773102eb
2020-09-07 17:17:59,322 analyzer.py:main@149 [INFO] maxexecs_addr 77341308
2020-09-07 17:17:59,322 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,322 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:17:59,322 analyzer.py:main@177 [INFO] <----------Mixing------------->
2020-09-07 17:17:59,322 analyzer.py:main@183 [INFO] <----------Shifting------------->
2020-09-07 17:17:59,322 analyzer.py:main@148 [INFO] entrypoint 773132cf
2020-09-07 17:17:59,322 analyzer.py:main@149 [INFO] maxexecs_addr 77313339
2020-09-07 17:17:59,322 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,322 analyzer.py:main@177 [INFO] <----------Mixing------------->
2020-09-07 17:17:59,323 analyzer.py:main@148 [INFO] entrypoint 773133a9
2020-09-07 17:17:59,323 analyzer.py:main@149 [INFO] maxexecs_addr 773133f8
2020-09-07 17:17:59,323 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,323 analyzer.py:main@148 [INFO] entrypoint 77313f64
2020-09-07 17:17:59,323 analyzer.py:main@149 [INFO] maxexecs_addr 773141e1
2020-09-07 17:17:59,323 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,323 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:17:59,323 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:17:59,323 analyzer.py:main@177 [INFO] <----------Mixing------------->
2020-09-07 17:17:59,323 analyzer.py:main@148 [INFO] entrypoint 77315f2c
2020-09-07 17:17:59,323 analyzer.py:main@149 [INFO] maxexecs_addr 773170ff
2020-09-07 17:17:59,323 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,323 analyzer.py:main@177 [INFO] <----------Mixing------------->
2020-09-07 17:17:59,324 analyzer.py:main@148 [INFO] entrypoint 77316d97
2020-09-07 17:17:59,324 analyzer.py:main@149 [INFO] maxexecs_addr 77316e52
2020-09-07 17:17:59,324 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,324 analyzer.py:main@148 [INFO] entrypoint 773175e6
```

```
2020-09-07 17:17:59,324 analyzer.py:main@149 [INFO] maxexecs_addr 77317652
2020-09-07 17:17:59,324 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,324 analyzer.py:main@148 [INFO] entrypoint 7731cb20
2020-09-07 17:17:59,324 analyzer.py:main@149 [INFO] maxexecs_addr 7731cb8a
2020-09-07 17:17:59,324 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,324 analyzer.py:main@148 [INFO] entrypoint 77324066
2020-09-07 17:17:59,324 analyzer.py:main@149 [INFO] maxexecs_addr 77341a52
2020-09-07 17:17:59,324 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,324 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:17:59,324 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 17:17:59,325 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:17:59,325 analyzer.py:main@183 [INFO] <----------Shifting------------>
2020-09-07 17:17:59,325 analyzer.py:main@148 [INFO] entrypoint 773cf51a
2020-09-07 17:17:59,325 analyzer.py:main@149 [INFO] maxexecs_addr 773cf555
2020-09-07 17:17:59,325 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,325 analyzer.py:main@148 [INFO] entrypoint 773e4168
2020-09-07 17:17:59,325 analyzer.py:main@149 [INFO] maxexecs_addr 773e4183
2020-09-07 17:17:59,325 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,325 analyzer.py:main@148 [INFO] entrypoint 773e5340
2020-09-07 17:17:59,325 analyzer.py:main@149 [INFO] maxexecs_addr 773e5385
2020-09-07 17:17:59,325 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:17:59,325 analyzer.py:main@148 [INFO] entrypoint 773e5bd0
2020-09-07 17:17:59,325 analyzer.py:main@149 [INFO] maxexecs_addr 773e5be0
2020-09-07 17:17:59,325 analyzer.py:main@148 [INFO] entrypoint 77402c6a
2020-09-07 17:17:59,325 analyzer.py:main@149 [INFO] maxexecs_addr 77402c6a
2020-09-07 17:17:59,326 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,326 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:17:59,326 analyzer.py:main@148 [INFO] entrypoint 77402dd6
2020-09-07 17:17:59,326 analyzer.py:main@149 [INFO] maxexecs_addr 77402dd6
2020-09-07 17:17:59,326 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,326 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:17:59,326 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 17:17:59,326 analyzer.py:main@148 [INFO] entrypoint 774058b5
2020-09-07 17:17:59,347 analyzer.py:main@149 [INFO] maxexecs_addr 774058d4
2020-09-07 17:17:59,347 analyzer.py:main@148 [INFO] entrypoint 77405aec
2020-09-07 17:17:59,347 analyzer.py:main@149 [INFO] maxexecs_addr 773de803
2020-09-07 17:17:59,347 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,349 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:17:59,349 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 17:17:59,349 analyzer.py:main@148 [INFO] entrypoint 7740617c
2020-09-07 17:17:59,349 analyzer.py:main@149 [INFO] maxexecs_addr 773cfb9c
2020-09-07 17:17:59,349 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,349 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:17:59,350 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 17:17:59,350 analyzer.py:main@148 [INFO] entrypoint 77406f0a
2020-09-07 17:17:59,350 analyzer.py:main@149 [INFO] maxexecs_addr 77406f30
2020-09-07 17:17:59,350 analyzer.py:main@148 [INFO] entrypoint 77407f9a
2020-09-07 17:17:59,350 analyzer.py:main@149 [INFO] maxexecs_addr 7740803b
2020-09-07 17:17:59,350 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,350 analyzer.py:main@148 [INFO] entrypoint 7740813b
2020-09-07 17:17:59,350 analyzer.py:main@149 [INFO] maxexecs_addr 77408323
2020-09-07 17:17:59,350 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,350 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:17:59,350 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 17:17:59,350 analyzer.py:main@148 [INFO] entrypoint 77408607
2020-09-07 17:17:59,350 analyzer.py:main@149 [INFO] maxexecs_addr 77408621
2020-09-07 17:17:59,350 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,350 analyzer.py:main@148 [INFO] entrypoint 7740898c
2020-09-07 17:17:59,351 analyzer.py:main@149 [INFO] maxexecs_addr 774089a1
2020-09-07 17:17:59,351 analyzer.py:main@148 [INFO] entrypoint 77408b19
2020-09-07 17:17:59,351 analyzer.py:main@149 [INFO] maxexecs_addr 77408c3c
2020-09-07 17:17:59,351 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,351 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 17:17:59,351 analyzer.py:main@148 [INFO] entrypoint 77408dac
2020-09-07 17:17:59,351 analyzer.py:main@149 [INFO] maxexecs_addr 77408df7
2020-09-07 17:17:59,351 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,351 analyzer.py:main@148 [INFO] entrypoint 77409bec
2020-09-07 17:17:59,351 analyzer.py:main@149 [INFO] maxexecs_addr 77409bec
2020-09-07 17:17:59,354 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,354 analyzer.py:main@148 [INFO] entrypoint 7740e3b2
2020-09-07 17:17:59,354 analyzer.py:main@149 [INFO] maxexecs_addr 7740e0d8
2020-09-07 17:17:59,354 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:17:59,354 analyzer.py:main@148 [INFO] entrypoint 7740fc8d
```

```
2020-09-07 17:17:59,355 analyzer.py:main@149 [INFO] maxexecs_addr 7740fd6d
2020-09-07 17:17:59,355 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,355 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:17:59,355 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:17:59,355 analyzer.py:main@148 [INFO] entrypoint 7740fd97
2020-09-07 17:17:59,355 analyzer.py:main@149 [INFO] maxexecs_addr 7740fdeb
2020-09-07 17:17:59,355 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,355 analyzer.py:main@148 [INFO] entrypoint 77414a61
2020-09-07 17:17:59,355 analyzer.py:main@149 [INFO] maxexecs_addr 77414a8c
2020-09-07 17:17:59,355 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,355 analyzer.py:main@148 [INFO] entrypoint 7741ff51
2020-09-07 17:17:59,355 analyzer.py:main@149 [INFO] maxexecs_addr 7741ff51
2020-09-07 17:17:59,355 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,355 analyzer.py:main@148 [INFO] entrypoint 7741fff1
2020-09-07 17:17:59,355 analyzer.py:main@149 [INFO] maxexecs_addr 773dadd1
2020-09-07 17:17:59,356 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:17:59,356 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:17:59,356 analyzer.py:main@190 [INFO] Collecting state bases
2020-09-07 17:17:59,357 analyzer.py:main@211 [INFO] Report overall stats
2020-09-07 17:17:59,357 analyzer.py:main@212 [INFO] entrypoint of comprehended caller 76351fb7
2020-09-07 17:17:59,357 analyzer.py:main@223 [INFO] Collect & report routines/functions, max 3
     if there are duplicates
2020-09-07 17:17:59,365 analyzer.py:main@264 [INFO] Overall routine reporting
2020-09-07 17:17:59,368 analyzer.py:main@269 [INFO] Round routine is: 404d58 ..)
2020-09-07 17:17:59,368 analyzer.py:main@274 [INFO] Main function is: 402094 ..)
2020-09-07 17:17:59,368 analyzer.py:main@280 [INFO] Collecting tap points data: states (
    ciphertexts or plaintexts)
2020-09-07 17:17:59,369 analyzer.py:main@295 [INFO] Collecting tap points data: key(s)
2020-09-07 17:17:59,370 analyzer.py:main@304 [INFO] Writing tap files (if any)
2020-09-07 17:17:59,370 analyzer.py:main@319 [INFO] Analyzer finished analyzing trace of
    112373006 bytes at 2020-09-07 17:17:59.370810
Duration (excl. verifier): 11 minute(s) for REPLAY serpent128 and ASID 31ce0000 and CALLER
    7740a5f0
Running CipherTrace's Verifier Light for REPLAY serpent128 and ASID 31ce0000 with CipherTrace
    Analyzer's output and stringsearch
2020-09-07 17:17:59,906 verifierlight.py:main@9 [INFO] Starting Verifier light at 2020-09-07
    17:17:59.906385
2020-09-07 17:17:59,909 verifierlight.py:main@12 [INFO] Reading matches file
    enc_string_matches.txt
2020-09-07 17:17:59,909 verifierlight.py:main@17 [INFO] Reading .tap files in directory file .
2020-09-07 17:17:59,909 verifierlight.py:main@23 [INFO] Reading tap file ./
    serpent128_76351fb7_states.tap
2020-09-07 17:17:59,910 verifierlight.py:main@23 [INFO] Reading tap file ./
    serpent128_76351fb7_keys.tap
2020-09-07 17:17:59,911 verifierlight.py:main@36 [DEBUG] Callers found: defaultdict(<type 'int
    '>, {'4021da': 11, '773140eb': 2, '401236': 104, '401c27': 1, '773176f8': 4})
2020-09-07 17:17:59,911 verifierlight.py:main@37 [DEBUG] PCs found: defaultdict(<type 'int'>,
    {'4023aa': 1, '40244b': 1})
2020-09-07 17:17:59,911 verifierlight.py:main@39 [INFO] The verifier should find data series
2020-09-07 17:17:59,911 verifierlight.py:main@42 [INFO] Verifier light finished searching PCs
    and Callers in 27 matches at 2020-09-07 17:17:59.911805
[omitted]
Running CipherTrace's Verifier with textprinter output
[omitted]
Running textprinter for TAP serpent128_76351fb7_states.tap and REPLAY serpent128 and ASID 31
    ce0000 with CipherTrace Analyzer's output
[omitted]
2020-09-07 17:20:51,019 verifier.py:main@67 [INFO] Data Series 1 was found in the write tap
    buffers, starting from line 4 with till 15
2020-09-07 17:20:51,019 verifier.py:main@63 [INFO] Data Series 3 was found in the read tap
    buffers, starting from line 4 till 20
2020-09-07 17:20:51,019 verifier.py:main@68 [INFO] Verifier finished searching 3 lines in
    buffers at 2020-09-07 17:20:51.019357
Duration: 14 minute(s) for REPLAY serpent128 and ASID 31ce0000 and Caller 7740a5f0
[omitted]
Duration: 25 minute(s) for REPLAY serpent128 and ASID 31ce0000 and Caller 401c97
Duration: 25 minute(s) for REPLAY serpent128 and ASID 31ce0000
[omitted]
```

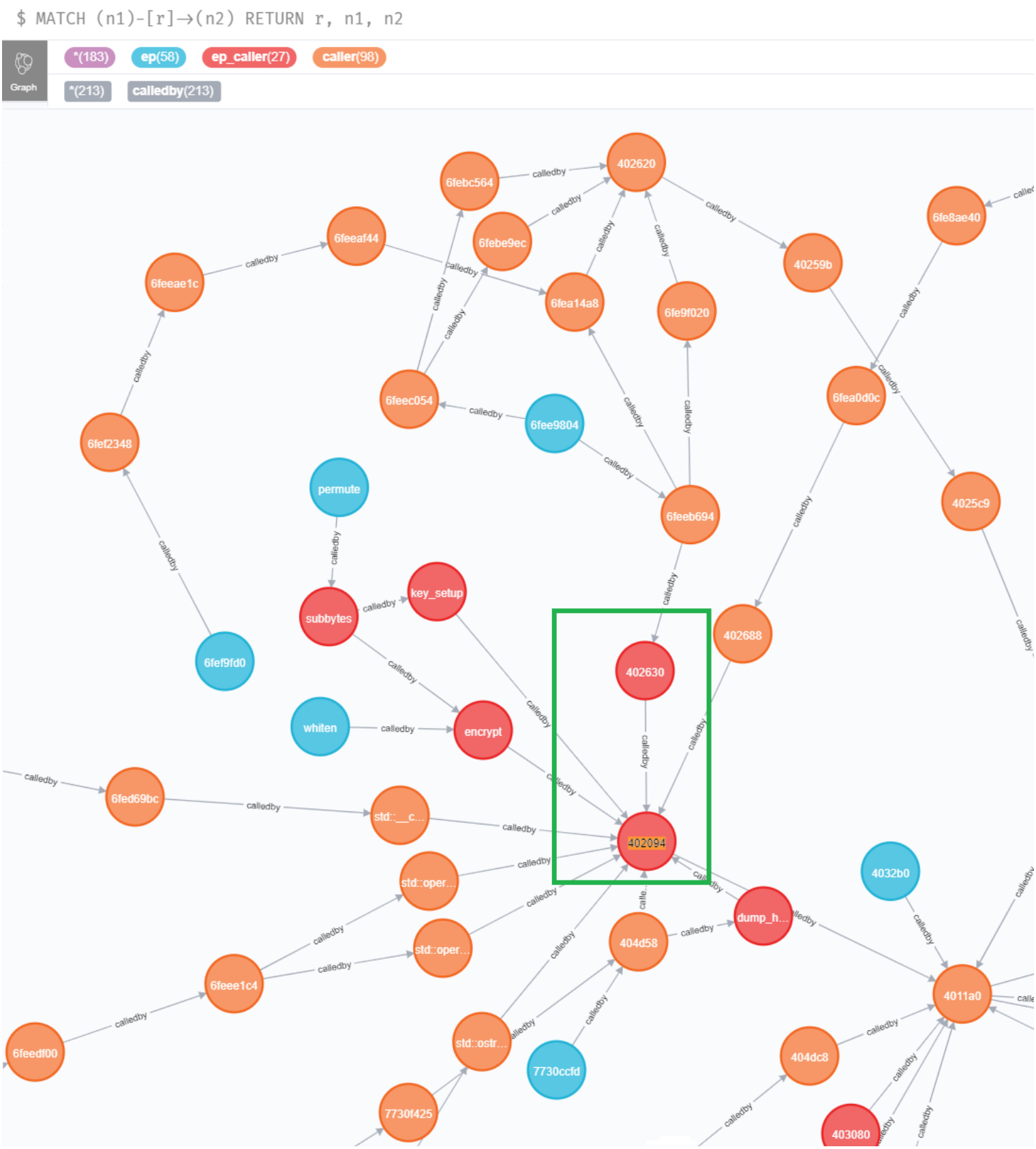

Figure A.4: SERPENT256 CFG

*The biggest shaped one is the finding.
*The highlighted one is the main function of the finding.

TF128

```
[omitted]
Running CipherTrace's Randometer with unigrams output
[omitted]
2020-09-07 17:53:04,306 randometer.py:main@45 [INFO] Starting Randometer for information
    measurement at 2020-09-07 17:53:04.306368
2020-09-07 17:53:04,307 randometer.py:main@46 [INFO] Reading unigram read file
    unigram_mem_read_report.bin
2020-09-07 17:53:05,091 randometer.py:main@52 [INFO] Reading unigram write file
    unigram_mem_write_report.bin
2020-09-07 17:53:05,600 randometer.py:main@60 [INFO] Computing randomness of read buffers
    using Chi-Squared test...
2020-09-07 17:53:05,644 randometer.py:main@63 [INFO] Computing randomness of write buffers
    using Chi-Squared test...
2020-09-07 17:53:05,675 randometer.py:main@68 [INFO] Computing read buffer entropy...
2020-09-07 17:53:05,795 randometer.py:main@71 [INFO] Computing write buffer entropy...
2020-09-07 17:53:05,872 randometer.py:main@74 [INFO] Entropy reads: 18692 writes: 12402
2020-09-07 17:53:05,872 randometer.py:main@77 [INFO] Applying read entropy mask > 0:
2020-09-07 17:53:05,875 randometer.py:main@81 [INFO] Applying write entropy mask > 0:
2020-09-07 17:53:05,876 randometer.py:main@87 [INFO] Applying read rand mask > 10000:
2020-09-07 17:53:05,885 randometer.py:main@91 [INFO] Applying write rand mask < 1000:
2020-09-07 17:53:05,945 randometer.py:main@105 [INFO] Results: reads: 946, writes: 254
[omitted]
Running CipherTrace's Analyzer for CALLER 402790 and REPLAY tf256 and ASID 30a98000 with
    func_stats output
2020-09-07 17:54:07,470 analyzer.py:main@15 [INFO] Starting Analyzer for a 133623169 bytes
    trace func_stats at 2020-09-07 17:54:07.470616
2020-09-07 17:54:07,471 analyzer.py:main@21 [INFO] filtering criteria from config: maxexecs,
    3, 1.0
2020-09-07 17:54:07,471 analyzer.py:main@34 [INFO] Reading the symbols file func_db
2020-09-07 17:54:07,473 analyzer.py:main@40 [INFO] Reading the stats file func_stats
2020-09-07 17:54:13,394 analyzer.py:main@50 [INFO] Filtering by caller
2020-09-07 17:54:13,491 analyzer.py:main@52 [INFO] filteredByCaller count: 1
2020-09-07 17:54:13,491 analyzer.py:main@54 [INFO] Collect the records with fields (maxexecs,
    llvm_bb) that have maximum values
2020-09-07 17:54:13,492 analyzer.py:main@57 [INFO] Find the main stack records (where it all
    begins, with max values of the maxexecs)
2020-09-07 17:54:13,492 analyzer.py:main@60 [INFO] mainRecs count: 1
2020-09-07 17:54:13,492 analyzer.py:main@66 [INFO] Exclude the main records from the filtered
    by caller ones be able to apply stack filtering.
2020-09-07 17:54:13,492 analyzer.py:main@73 [INFO] Do the analysis for each comprehended
    caller stack
Duration (excl. verifier): 7 minute(s) for REPLAY tf256 and ASID 30a98000 and CALLER 402790
There are no tap files to check for caller 402790 and replay tf256
Duration: 7 minute(s) for REPLAY tf256 and ASID 30a98000 and Caller 402790
Running CipherTrace's Analyzer for CALLER 401af8 and REPLAY tf256 and ASID 30a98000 with
    func_stats output
2020-09-07 17:54:14,102 analyzer.py:main@15 [INFO] Starting Analyzer for a 133623169 bytes
    trace func_stats at 2020-09-07 17:54:14.102123
2020-09-07 17:54:14,102 analyzer.py:main@21 [INFO] filtering criteria from config: maxexecs,
    3, 1.0
2020-09-07 17:54:14,102 analyzer.py:main@34 [INFO] Reading the symbols file func_db
2020-09-07 17:54:14,103 analyzer.py:main@40 [INFO] Reading the stats file func_stats
2020-09-07 17:54:19,925 analyzer.py:main@50 [INFO] Filtering by caller
2020-09-07 17:54:20,029 analyzer.py:main@52 [INFO] filteredByCaller count: 32
2020-09-07 17:54:20,029 analyzer.py:main@54 [INFO] Collect the records with fields (maxexecs,
    llvm_bb) that have maximum values
2020-09-07 17:54:20,030 analyzer.py:main@57 [INFO] Find the main stack records (where it all
    begins, with max values of the maxexecs)
2020-09-07 17:54:20,030 analyzer.py:main@60 [INFO] mainRecs count: 7
2020-09-07 17:54:20,030 analyzer.py:main@66 [INFO] Exclude the main records from the filtered
    by caller ones be able to apply stack filtering.
2020-09-07 17:54:20,030 analyzer.py:main@73 [INFO] Do the analysis for each comprehended
    caller stack
2020-09-07 17:54:20,030 analyzer.py:main@97 [INFO] Stack name: functionstack
2020-09-07 17:54:20,030 analyzer.py:main@98 [INFO] Stack filter: [u'828a4c52', u'82897534', u
    '773f5568']
2020-09-07 17:54:20,031 analyzer.py:main@101 [INFO] Filter the records: those which share the
    stack filter, and certain stats (arith>1, loop>1, compc_write_entropy>1.0)
2020-09-07 17:54:20,222 analyzer.py:main@107 [INFO] Create the graph file (for CFG)
2020-09-07 17:54:20,404 analyzer.py:main@116 [INFO] Aggregate by maxexecs and groupby the
    entrypoint (function name)
2020-09-07 17:54:20,410 analyzer.py:main@120 [INFO] Find the crypto elements, as per to
    certain traits
2020-09-07 17:54:22,123 analyzer.py:main@148 [INFO] entrypoint 401410
2020-09-07 17:54:22,123 analyzer.py:main@149 [INFO] maxexecs_addr 4015b4
2020-09-07 17:54:22,123 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:54:22,124 analyzer.py:main@148 [INFO] entrypoint 40167e
2020-09-07 17:54:22,124 analyzer.py:main@149 [INFO] maxexecs_addr 4016ce
2020-09-07 17:54:22,124 analyzer.py:main@148 [INFO] entrypoint 401810
```

```
2020-09-07 17:54:22,124 analyzer.py:main@148 [INFO] entrypoint 401b5e
2020-09-07 17:54:22,125 analyzer.py:main@149 [INFO] maxexecs_addr 401b6b
2020-09-07 17:54:22,125 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,125 analyzer.py:main@148 [INFO] entrypoint 401c8e
2020-09-07 17:54:22,125 analyzer.py:main@149 [INFO] maxexecs_addr 401d65
2020-09-07 17:54:22,125 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,125 analyzer.py:main@148 [INFO] entrypoint 401e02
2020-09-07 17:54:22,125 analyzer.py:main@149 [INFO] maxexecs_addr 401e02
2020-09-07 17:54:22,125 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,125 analyzer.py:main@148 [INFO] entrypoint 402248
2020-09-07 17:54:22,125 analyzer.py:main@149 [INFO] maxexecs_addr 402276
2020-09-07 17:54:22,126 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,126 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:54:22,126 analyzer.py:main@183 [INFO] <----------Shifting------------>
2020-09-07 17:54:22,126 analyzer.py:main@148 [INFO] entrypoint 40279e
2020-09-07 17:54:22,126 analyzer.py:main@149 [INFO] maxexecs_addr 402823
2020-09-07 17:54:22,126 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,126 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:54:22,126 analyzer.py:main@183 [INFO] <----------Shifting------------>
2020-09-07 17:54:22,127 analyzer.py:main@148 [INFO] entrypoint 403bde
2020-09-07 17:54:22,127 analyzer.py:main@149 [INFO] maxexecs_addr 403ffa
2020-09-07 17:54:22,127 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,127 analyzer.py:main@148 [INFO] entrypoint 404da1
2020-09-07 17:54:22,127 analyzer.py:main@149 [INFO] maxexecs_addr 404dae
2020-09-07 17:54:22,127 analyzer.py:main@148 [INFO] entrypoint 4050a0
2020-09-07 17:54:22,127 analyzer.py:main@149 [INFO] maxexecs_addr 405123
2020-09-07 17:54:22,127 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,127 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:54:22,127 analyzer.py:main@148 [INFO] entrypoint 4055b0
2020-09-07 17:54:22,128 analyzer.py:main@149 [INFO] maxexecs_addr 40557a
2020-09-07 17:54:22,128 analyzer.py:main@148 [INFO] entrypoint 405970
2020-09-07 17:54:22,128 analyzer.py:main@149 [INFO] maxexecs_addr 405a01
2020-09-07 17:54:22,128 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,128 analyzer.py:main@148 [INFO] entrypoint 405a60
2020-09-07 17:54:22,128 analyzer.py:main@149 [INFO] maxexecs_addr 405ace
2020-09-07 17:54:22,128 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:54:22,128 analyzer.py:main@148 [INFO] entrypoint 405c58
2020-09-07 17:54:22,128 analyzer.py:main@149 [INFO] maxexecs_addr 82878337
2020-09-07 17:54:22,128 analyzer.py:main@148 [INFO] entrypoint 406040
2020-09-07 17:54:22,128 analyzer.py:main@149 [INFO] maxexecs_addr 406068
2020-09-07 17:54:22,128 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,128 analyzer.py:main@148 [INFO] entrypoint 406250
2020-09-07 17:54:22,128 analyzer.py:main@149 [INFO] maxexecs_addr 40626a
2020-09-07 17:54:22,128 analyzer.py:main@148 [INFO] entrypoint 4063e0
2020-09-07 17:54:22,129 analyzer.py:main@149 [INFO] maxexecs_addr 406685
2020-09-07 17:54:22,129 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,129 analyzer.py:main@148 [INFO] entrypoint 406da0
2020-09-07 17:54:22,129 analyzer.py:main@149 [INFO] maxexecs_addr 406f28
2020-09-07 17:54:22,129 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,129 analyzer.py:main@148 [INFO] entrypoint 407270
2020-09-07 17:54:22,129 analyzer.py:main@149 [INFO] maxexecs_addr 4072a0
2020-09-07 17:54:22,129 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,129 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:54:22,129 analyzer.py:main@148 [INFO] entrypoint 407320
2020-09-07 17:54:22,129 analyzer.py:main@149 [INFO] maxexecs_addr 407363
2020-09-07 17:54:22,129 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,129 analyzer.py:main@148 [INFO] entrypoint 407700
2020-09-07 17:54:22,129 analyzer.py:main@149 [INFO] maxexecs_addr 77315381
2020-09-07 17:54:22,129 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,129 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:54:22,129 analyzer.py:main@148 [INFO] entrypoint 407718
2020-09-07 17:54:22,130 analyzer.py:main@149 [INFO] maxexecs_addr 77309cc6
2020-09-07 17:54:22,130 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,130 analyzer.py:main@148 [INFO] entrypoint 407720
2020-09-07 17:54:22,130 analyzer.py:main@149 [INFO] maxexecs_addr 77309b60
2020-09-07 17:54:22,130 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,130 analyzer.py:main@148 [INFO] entrypoint 407728
2020-09-07 17:54:22,130 analyzer.py:main@149 [INFO] maxexecs_addr 773179b0
2020-09-07 17:54:22,130 analyzer.py:main@148 [INFO] entrypoint 4077b8
2020-09-07 17:54:22,130 analyzer.py:main@149 [INFO] maxexecs_addr 76353d6b
2020-09-07 17:54:22,130 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,130 analyzer.py:main@148 [INFO] entrypoint 407800
2020-09-07 17:54:22,130 analyzer.py:main@149 [INFO] maxexecs_addr 82878337
```

```
2020-09-07 17:54:22,130 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,130 analyzer.py:main@148 [INFO] entrypoint 407838
2020-09-07 17:54:22,130 analyzer.py:main@149 [INFO] maxexecs_addr 773247c3
2020-09-07 17:54:22,130 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,131 analyzer.py:main@148 [INFO] entrypoint 6eb41000
2020-09-07 17:54:22,131 analyzer.py:main@149 [INFO] maxexecs_addr 6eb41028
2020-09-07 17:54:22,131 analyzer.py:main@148 [INFO] entrypoint 6fe41000
2020-09-07 17:54:22,131 analyzer.py:main@149 [INFO] maxexecs_addr 6fe41028
2020-09-07 17:54:22,131 analyzer.py:main@148 [INFO] entrypoint 6fe6dda0
2020-09-07 17:54:22,131 analyzer.py:main@149 [INFO] maxexecs_addr 6fe6ddb7
2020-09-07 17:54:22,131 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,131 analyzer.py:main@148 [INFO] entrypoint 6fea0244
2020-09-07 17:54:22,131 analyzer.py:main@149 [INFO] maxexecs_addr 82878337
2020-09-07 17:54:22,131 analyzer.py:main@148 [INFO] entrypoint 6fed7fd8
2020-09-07 17:54:22,131 analyzer.py:main@149 [INFO] maxexecs_addr 82878337
2020-09-07 17:54:22,131 analyzer.py:main@148 [INFO] entrypoint 6fee9804
2020-09-07 17:54:22,131 analyzer.py:main@149 [INFO] maxexecs_addr 6fee986f
2020-09-07 17:54:22,131 analyzer.py:main@148 [INFO] entrypoint 75766788
2020-09-07 17:54:22,131 analyzer.py:main@149 [INFO] maxexecs_addr 773e4cf3
2020-09-07 17:54:22,131 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,132 analyzer.py:main@148 [INFO] entrypoint 757680b0
2020-09-07 17:54:22,132 analyzer.py:main@149 [INFO] maxexecs_addr 757681bc
2020-09-07 17:54:22,132 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,132 analyzer.py:main@148 [INFO] entrypoint 7577f7b7
2020-09-07 17:54:22,132 analyzer.py:main@149 [INFO] maxexecs_addr 7577f7f8
2020-09-07 17:54:22,132 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,132 analyzer.py:main@148 [INFO] entrypoint 77309bfd
2020-09-07 17:54:22,132 analyzer.py:main@149 [INFO] maxexecs_addr 77309c56
2020-09-07 17:54:22,132 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,132 analyzer.py:main@148 [INFO] entrypoint 77309c67
2020-09-07 17:54:22,132 analyzer.py:main@149 [INFO] maxexecs_addr 77309c7f
2020-09-07 17:54:22,132 analyzer.py:main@148 [INFO] entrypoint 7730a544
2020-09-07 17:54:22,132 analyzer.py:main@149 [INFO] maxexecs_addr 7730a58f
2020-09-07 17:54:22,132 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,132 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:54:22,132 analyzer.py:main@148 [INFO] entrypoint 7730a7c3
2020-09-07 17:54:22,133 analyzer.py:main@149 [INFO] maxexecs_addr 7730a824
2020-09-07 17:54:22,133 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,133 analyzer.py:main@148 [INFO] entrypoint 7730ae77
2020-09-07 17:54:22,133 analyzer.py:main@149 [INFO] maxexecs_addr 7730aed6
2020-09-07 17:54:22,133 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,133 analyzer.py:main@148 [INFO] entrypoint 7730ccfd
2020-09-07 17:54:22,133 analyzer.py:main@149 [INFO] maxexecs_addr 7730ce4e
2020-09-07 17:54:22,133 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,133 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 17:54:22,133 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 17:54:22,133 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:54:22,133 analyzer.py:main@183 [INFO] <----------Shifting------------>
2020-09-07 17:54:22,133 analyzer.py:main@148 [INFO] entrypoint 7730f574
2020-09-07 17:54:22,133 analyzer.py:main@149 [INFO] maxexecs_addr 7730f5a1
2020-09-07 17:54:22,133 analyzer.py:main@148 [INFO] entrypoint 773102eb
2020-09-07 17:54:22,133 analyzer.py:main@149 [INFO] maxexecs_addr 77341308
2020-09-07 17:54:22,134 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,134 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 17:54:22,134 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:54:22,136 analyzer.py:main@183 [INFO] <----------Shifting------------>
2020-09-07 17:54:22,136 analyzer.py:main@148 [INFO] entrypoint 773132cf
2020-09-07 17:54:22,136 analyzer.py:main@149 [INFO] maxexecs_addr 77313339
2020-09-07 17:54:22,136 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,136 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:54:22,136 analyzer.py:main@148 [INFO] entrypoint 77313f64
2020-09-07 17:54:22,137 analyzer.py:main@149 [INFO] maxexecs_addr 773141e1
2020-09-07 17:54:22,137 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,137 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:54:22,137 analyzer.py:main@148 [INFO] entrypoint 77315f2c
2020-09-07 17:54:22,137 analyzer.py:main@149 [INFO] maxexecs_addr 773170ff
2020-09-07 17:54:22,137 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,137 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 17:54:22,137 analyzer.py:main@148 [INFO] entrypoint 7731cb20
2020-09-07 17:54:22,137 analyzer.py:main@149 [INFO] maxexecs_addr 7731cb8a
2020-09-07 17:54:22,137 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 17:54:22,137 analyzer.py:main@148 [INFO] entrypoint 77324066
2020-09-07 17:54:22,137 analyzer.py:main@149 [INFO] maxexecs_addr 77341a52
```

```
2020-09-07 17:54:22,137 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:54:22,137 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:54:22,137 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:54:22,137 analyzer.py:main@177 [INFO] <----------Mixing------------->
2020-09-07 17:54:22,137 analyzer.py:main@183 [INFO] <----------Shifting------------->
2020-09-07 17:54:22,138 analyzer.py:main@148 [INFO] entrypoint 773cf51a
2020-09-07 17:54:22,138 analyzer.py:main@149 [INFO] maxexecs_addr 773cf555
2020-09-07 17:54:22,138 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:54:22,139 analyzer.py:main@148 [INFO] entrypoint 773e4168
2020-09-07 17:54:22,139 analyzer.py:main@149 [INFO] maxexecs_addr 773e4183
2020-09-07 17:54:22,139 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:54:22,140 analyzer.py:main@148 [INFO] entrypoint 773e5bd0
2020-09-07 17:54:22,140 analyzer.py:main@149 [INFO] maxexecs_addr 773e5be0
2020-09-07 17:54:22,140 analyzer.py:main@148 [INFO] entrypoint 77402c6a
2020-09-07 17:54:22,140 analyzer.py:main@149 [INFO] maxexecs_addr 77402c6a
2020-09-07 17:54:22,140 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:54:22,140 analyzer.py:main@148 [INFO] entrypoint 77402dd6
2020-09-07 17:54:22,140 analyzer.py:main@149 [INFO] maxexecs_addr 77402dd6
2020-09-07 17:54:22,140 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:54:22,140 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:54:22,140 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:54:22,140 analyzer.py:main@148 [INFO] entrypoint 774058b5
2020-09-07 17:54:22,141 analyzer.py:main@149 [INFO] maxexecs_addr 774058d4
2020-09-07 17:54:22,141 analyzer.py:main@148 [INFO] entrypoint 77405aec
2020-09-07 17:54:22,141 analyzer.py:main@149 [INFO] maxexecs_addr 773de803
2020-09-07 17:54:22,141 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:54:22,141 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:54:22,141 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:54:22,141 analyzer.py:main@148 [INFO] entrypoint 7740617c
2020-09-07 17:54:22,141 analyzer.py:main@149 [INFO] maxexecs_addr 773cfb9c
2020-09-07 17:54:22,141 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:54:22,141 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:54:22,141 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:54:22,142 analyzer.py:main@148 [INFO] entrypoint 77406f0a
2020-09-07 17:54:22,142 analyzer.py:main@149 [INFO] maxexecs_addr 77406f30
2020-09-07 17:54:22,142 analyzer.py:main@148 [INFO] entrypoint 77409bec
2020-09-07 17:54:22,142 analyzer.py:main@149 [INFO] maxexecs_addr 77409bec
2020-09-07 17:54:22,142 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:54:22,142 analyzer.py:main@148 [INFO] entrypoint 7740e3b2
2020-09-07 17:54:22,142 analyzer.py:main@149 [INFO] maxexecs_addr 7740e0d8
2020-09-07 17:54:22,142 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:54:22,142 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:54:22,142 analyzer.py:main@148 [INFO] entrypoint 7740fc8d
2020-09-07 17:54:22,142 analyzer.py:main@149 [INFO] maxexecs_addr 7740fd6d
2020-09-07 17:54:22,142 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:54:22,142 analyzer.py:main@159 [INFO] <----------KRoundInit------------->
2020-09-07 17:54:22,142 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:54:22,143 analyzer.py:main@148 [INFO] entrypoint 77414a61
2020-09-07 17:54:22,143 analyzer.py:main@149 [INFO] maxexecs_addr 77414a8c
2020-09-07 17:54:22,150 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:54:22,158 analyzer.py:main@148 [INFO] entrypoint 7741ff51
2020-09-07 17:54:22,158 analyzer.py:main@149 [INFO] maxexecs_addr 7741ff51
2020-09-07 17:54:22,158 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:54:22,158 analyzer.py:main@148 [INFO] entrypoint 7741fff1
2020-09-07 17:54:22,158 analyzer.py:main@149 [INFO] maxexecs_addr 773dadd1
2020-09-07 17:54:22,158 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 17:54:22,158 analyzer.py:main@165 [INFO] <----------KRound------------->
2020-09-07 17:54:22,158 analyzer.py:main@190 [INFO] Collecting state bases
2020-09-07 17:54:22,160 analyzer.py:main@211 [INFO] Report overall stats
2020-09-07 17:54:22,162 analyzer.py:main@212 [INFO] entrypoint of comprehended caller 40279e
2020-09-07 17:54:22,162 analyzer.py:main@223 [INFO] Collect & report routines/functions, max 3
     if there are duplicates
2020-09-07 17:54:22,171 analyzer.py:main@264 [INFO] Overall routine reporting
2020-09-07 17:54:22,173 analyzer.py:main@269 [INFO] Round routine is: Twofish::PrepareKey(
    uchar ..)
2020-09-07 17:54:22,176 analyzer.py:main@274 [INFO] Main function is: Twofish::test_sequence(
    int,uchar ..)
2020-09-07 17:54:22,176 analyzer.py:main@280 [INFO] Collecting tap points data: states (
    ciphertexts or plaintexts)
2020-09-07 17:54:22,179 analyzer.py:main@295 [INFO] Collecting tap points data: key(s)
2020-09-07 17:54:22,182 analyzer.py:main@304 [INFO] Writing tap files (if any)
2020-09-07 17:54:22,183 analyzer.py:main@319 [INFO] Analyzer finished analyzing trace of
    133623169 bytes at 2020-09-07 17:54:22.183267
```

```
Duration (excl. verifier): 7 minute(s) for REPLAY tf256 and ASID 30a98000 and CALLER 401af8
Running CipherTrace's Verifier Light for REPLAY tf256 and ASID 30a98000 with CipherTrace
    Analyzer's output and stringsearch
2020-09-07 17:54:22,854 verifierlight.py:main@9 [INFO] Starting Verifier light at 2020-09-07
    17:54:22.854130
2020-09-07 17:54:22,854 verifierlight.py:main@12 [INFO] Reading matches file
    enc_string_matches.txt
2020-09-07 17:54:22,854 verifierlight.py:main@17 [INFO] Reading .tap files in directory file .
2020-09-07 17:54:22,855 verifierlight.py:main@23 [INFO] Reading tap file ./tf256_40279e_keys.
    tap
2020-09-07 17:54:22,855 verifierlight.py:main@23 [INFO] Reading tap file ./tf256_40279e_states
    .tap
2020-09-07 17:54:22,858 verifierlight.py:main@36 [DEBUG] Callers found: defaultdict(<type 'int
    '>, {})
2020-09-07 17:54:22,858 verifierlight.py:main@37 [DEBUG] PCs found: defaultdict(<type 'int'>,
    {})
2020-09-07 17:54:22,862 verifierlight.py:main@41 [WARNING] The verifier may not find data
    series
2020-09-07 17:54:22,863 verifierlight.py:main@42 [INFO] Verifier light finished searching PCs
    and Callers in 3 matches at 2020-09-07 17:54:22.863287
Running textprinter for TAP tf256_40279e_keys.tap and REPLAY tf256 and ASID 30a98000 with
    CipherTrace Analyzer's output
[omitted]
Running CipherTrace's Verifier with textprinter output
[omitted]
2020-09-07 17:55:19,057 verifier.py:main@68 [INFO] Verifier finished searching 3 lines in
    buffers at 2020-09-07 17:55:19.057598
[omitted]
```

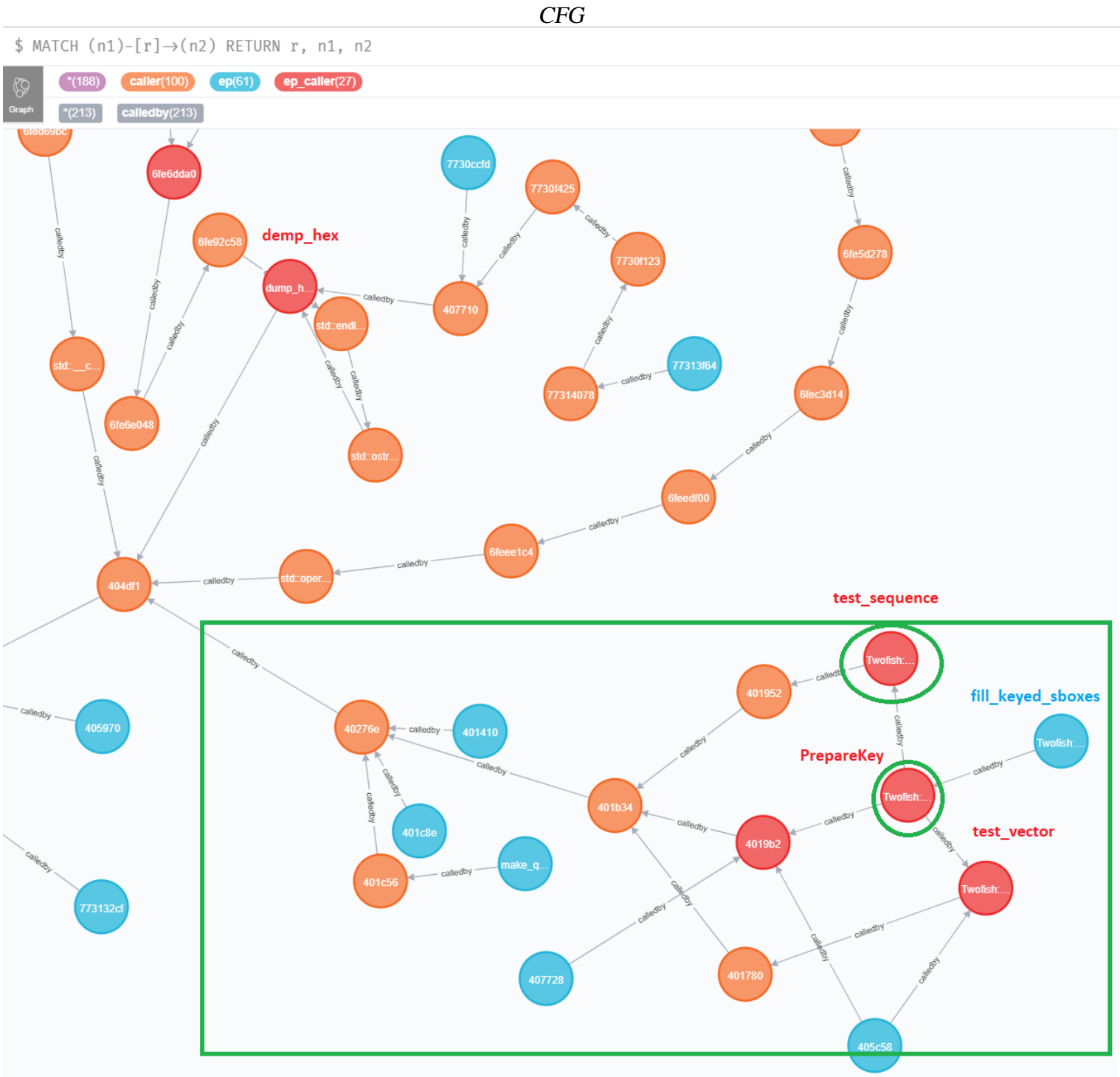

Figure A.5: Twofish128 CFG

*The biggest shaped one is the finding.
*The circles ones are the finding's main function and round routine.

OPENSSLAES256

```
[omitted]
Running CipherTrace's Randometer with unigrams output
2020-09-07 18:56:09,101 randometer.py:main@45 [INFO] Starting Randometer for information
    measurement at 2020-09-07 18:56:09.101801
2020-09-07 18:56:09,102 randometer.py:main@46 [INFO] Reading unigram read file
    unigram_mem_read_report.bin
2020-09-07 18:56:10,709 randometer.py:main@52 [INFO] Reading unigram write file
    unigram_mem_write_report.bin
2020-09-07 18:56:11,595 randometer.py:main@60 [INFO] Computing randomness of read buffers
    using Chi-Squared test...
2020-09-07 18:56:11,673 randometer.py:main@63 [INFO] Computing randomness of write buffers
    using Chi-Squared test...
2020-09-07 18:56:11,729 randometer.py:main@68 [INFO] Computing read buffer entropy...
2020-09-07 18:56:11,877 randometer.py:main@71 [INFO] Computing write buffer entropy...
2020-09-07 18:56:11,981 randometer.py:main@74 [INFO] Entropy reads: 23804 writes: 18566
2020-09-07 18:56:11,981 randometer.py:main@77 [INFO] Applying read entropy mask > 0:
2020-09-07 18:56:11,984 randometer.py:main@81 [INFO] Applying write entropy mask > 0:
2020-09-07 18:56:11,987 randometer.py:main@87 [INFO] Applying read rand mask > 10000:
2020-09-07 18:56:11,992 randometer.py:main@91 [INFO] Applying write rand mask < 1000:
2020-09-07 18:56:12,057 randometer.py:main@105 [INFO] Results: reads: 476, writes: 161
[omitted]
2020-09-07 18:56:12,155 randometer.py:main@140 [INFO] Callers for ASID 2ebe6000 are set(['6
    dd4df40', '6dd4ea51', '77405d5a', '74f87f7c', '74f88162', '74f87a8f', '74f8c017', '74
    f87f87', '362477', '7740a5f0', '74d23027', '74f8bfe8', '6dd440cb', '74f881f2', '6dd27e50',
     '74f88443', '74f87f3d'])
2020-09-07 18:56:12,155 randometer.py:main@142 [INFO] Randometer finished measuring
    information of unigram_mem_read_report.bin and unigram_mem_write_report.bin at 2020-09-07
    18:56:12.155917
[omitted]
Running CipherTrace's Analyzer for CALLER 77405d5a and REPLAY opensslaes256 and ASID 2ebe6000
    with func_stats output
[omitted]
Running CipherTrace's Analyzer for CALLER 77405d5a and REPLAY opensslaes256 and ASID 2ebe6000
    with func_stats output
2020-09-07 19:27:30,980 analyzer.py:main@15 [INFO] Starting Analyzer for a 638818822 bytes
    trace func_stats at 2020-09-07 19:27:30.980180
2020-09-07 19:27:30,980 analyzer.py:main@21 [INFO] filtering criteria from config: maxexecs,
    3, 1.0
2020-09-07 19:27:30,980 analyzer.py:main@34 [INFO] Reading the symbols file func_db
2020-09-07 19:27:31,015 analyzer.py:main@40 [INFO] Reading the stats file func_stats
2020-09-07 19:28:46,982 analyzer.py:main@50 [INFO] Filtering by caller
2020-09-07 19:30:01,046 analyzer.py:main@52 [INFO] filteredByCaller count: 4018
2020-09-07 19:30:01,050 analyzer.py:main@54 [INFO] Collect the records with fields (maxexecs,
    llvm_bb) that have maximum values
2020-09-07 19:30:01,585 analyzer.py:main@57 [INFO] Find the main stack records (where it all
    begins, with max values of the maxexecs)
2020-09-07 19:30:01,589 analyzer.py:main@60 [INFO] mainRecs count: 132
2020-09-07 19:30:01,590 analyzer.py:main@66 [INFO] Exclude the main records from the filtered
    by caller ones be able to apply stack filtering.
2020-09-07 19:30:04,799 analyzer.py:main@73 [INFO] Do the analysis for each comprehended
    caller stack
[omitted]
2020-09-07 19:55:18,246 analyzer.py:main@97 [INFO] Stack name: functionstack
2020-09-07 19:55:18,246 analyzer.py:main@98 [INFO] Stack filter: [u'74f89f4b', u'74f8a7cc', u
    '74f8a6a4']
2020-09-07 19:55:18,246 analyzer.py:main@101 [INFO] Filter the records: those which share the
    stack filter, and certain stats (arith>1, loop>1, compc_write_entropy>1.0)
2020-09-07 19:55:20,572 analyzer.py:main@107 [INFO] Create the graph file (for CFG)
2020-09-07 19:55:23,179 analyzer.py:main@116 [INFO] Aggregate by maxexecs and groupby the
    entrypoint (function name)
2020-09-07 19:55:23,260 analyzer.py:main@120 [INFO] Find the crypto elements, as per to
    certain traits
2020-09-07 19:55:30,521 analyzer.py:main@148 [INFO] entrypoint 74f87a40
2020-09-07 19:55:30,521 analyzer.py:main@149 [INFO] maxexecs_addr 74f87a8f
2020-09-07 19:55:30,522 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 19:55:30,524 analyzer.py:main@148 [INFO] entrypoint 74f87b90
2020-09-07 19:55:30,524 analyzer.py:main@149 [INFO] maxexecs_addr 74f87bd6
2020-09-07 19:55:30,524 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 19:55:30,524 analyzer.py:main@148 [INFO] entrypoint 74f87c20
2020-09-07 19:55:30,525 analyzer.py:main@149 [INFO] maxexecs_addr 74f87c40
2020-09-07 19:55:30,525 analyzer.py:main@148 [INFO] entrypoint 74f87c60
2020-09-07 19:55:30,525 analyzer.py:main@149 [INFO] maxexecs_addr 74f87f3d
2020-09-07 19:55:30,525 analyzer.py:main@153 [INFO] <----------Scheduling------------->
2020-09-07 19:55:30,526 analyzer.py:main@148 [INFO] entrypoint 74f88010
2020-09-07 19:55:30,526 analyzer.py:main@149 [INFO] maxexecs_addr 74f88023
2020-09-07 19:55:30,527 analyzer.py:main@148 [INFO] entrypoint 74f88040
2020-09-07 19:55:30,528 analyzer.py:main@149 [INFO] maxexecs_addr 74f88050
2020-09-07 19:55:30,529 analyzer.py:main@148 [INFO] entrypoint 74f88090
2020-09-07 19:55:30,530 analyzer.py:main@149 [INFO] maxexecs_addr 74f880b9
```

```
2020-09-07 19:55:30,531 analyzer.py:main@148 [INFO] entrypoint 74f88190
2020-09-07 19:55:30,534 analyzer.py:main@149 [INFO] maxexecs_addr 74f881f2
2020-09-07 19:55:30,535 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,536 analyzer.py:main@148 [INFO] entrypoint 74f882b0
2020-09-07 19:55:30,538 analyzer.py:main@149 [INFO] maxexecs_addr 74f8840b
2020-09-07 19:55:30,538 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,539 analyzer.py:main@148 [INFO] entrypoint 74f88510
2020-09-07 19:55:30,541 analyzer.py:main@149 [INFO] maxexecs_addr 74f8852e
2020-09-07 19:55:30,541 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 19:55:30,542 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 19:55:30,542 analyzer.py:main@171 [INFO] <----------S-Box------------>
2020-09-07 19:55:30,543 analyzer.py:main@148 [INFO] entrypoint 74f88554
2020-09-07 19:55:30,544 analyzer.py:main@149 [INFO] maxexecs_addr 74f88572
2020-09-07 19:55:30,544 analyzer.py:main@171 [INFO] <----------S-Box------------>
2020-09-07 19:55:30,544 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 19:55:30,545 analyzer.py:main@148 [INFO] entrypoint 74f885c0
2020-09-07 19:55:30,545 analyzer.py:main@149 [INFO] maxexecs_addr 74f88608
2020-09-07 19:55:30,546 analyzer.py:main@148 [INFO] entrypoint 74f88640
2020-09-07 19:55:30,555 analyzer.py:main@149 [INFO] maxexecs_addr 74f8869b
2020-09-07 19:55:30,555 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 19:55:30,555 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 19:55:30,556 analyzer.py:main@171 [INFO] <----------S-Box------------>
2020-09-07 19:55:30,558 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 19:55:30,559 analyzer.py:main@183 [INFO] <----------Shifting------------>
2020-09-07 19:55:30,559 analyzer.py:main@148 [INFO] entrypoint 74f88720
2020-09-07 19:55:30,559 analyzer.py:main@149 [INFO] maxexecs_addr 74f8877b
2020-09-07 19:55:30,559 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 19:55:30,559 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 19:55:30,559 analyzer.py:main@171 [INFO] <----------S-Box------------>
2020-09-07 19:55:30,559 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 19:55:30,559 analyzer.py:main@183 [INFO] <----------Shifting------------>
2020-09-07 19:55:30,559 analyzer.py:main@148 [INFO] entrypoint 74f88800
2020-09-07 19:55:30,559 analyzer.py:main@149 [INFO] maxexecs_addr 74f88813
2020-09-07 19:55:30,559 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,559 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 19:55:30,561 analyzer.py:main@148 [INFO] entrypoint 74f88afa
2020-09-07 19:55:30,561 analyzer.py:main@149 [INFO] maxexecs_addr 773e4cf3
2020-09-07 19:55:30,561 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,561 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 19:55:30,562 analyzer.py:main@148 [INFO] entrypoint 74f88b70
2020-09-07 19:55:30,562 analyzer.py:main@149 [INFO] maxexecs_addr 773e5385
2020-09-07 19:55:30,564 analyzer.py:main@148 [INFO] entrypoint 74f8940b
2020-09-07 19:55:30,564 analyzer.py:main@149 [INFO] maxexecs_addr 74f89485
2020-09-07 19:55:30,564 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,564 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 19:55:30,564 analyzer.py:main@148 [INFO] entrypoint 74f89999
2020-09-07 19:55:30,565 analyzer.py:main@149 [INFO] maxexecs_addr 74f899bf
2020-09-07 19:55:30,565 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,565 analyzer.py:main@148 [INFO] entrypoint 74f89a33
2020-09-07 19:55:30,566 analyzer.py:main@149 [INFO] maxexecs_addr 74f89a4c
2020-09-07 19:55:30,566 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,566 analyzer.py:main@148 [INFO] entrypoint 74f89b2a
2020-09-07 19:55:30,566 analyzer.py:main@149 [INFO] maxexecs_addr 74f89c2a
2020-09-07 19:55:30,566 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,566 analyzer.py:main@177 [INFO] <----------Mixing------------>
2020-09-07 19:55:30,567 analyzer.py:main@148 [INFO] entrypoint 74f89c80
2020-09-07 19:55:30,567 analyzer.py:main@149 [INFO] maxexecs_addr 74f89cba
2020-09-07 19:55:30,567 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,568 analyzer.py:main@148 [INFO] entrypoint 74f8a540
2020-09-07 19:55:30,568 analyzer.py:main@149 [INFO] maxexecs_addr 74f8a56b
2020-09-07 19:55:30,568 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,568 analyzer.py:main@148 [INFO] entrypoint 74f8a636
2020-09-07 19:55:30,569 analyzer.py:main@149 [INFO] maxexecs_addr 74f8a670
2020-09-07 19:55:30,569 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,570 analyzer.py:main@148 [INFO] entrypoint 74f8a6a4
2020-09-07 19:55:30,570 analyzer.py:main@149 [INFO] maxexecs_addr 74f8a6d9
2020-09-07 19:55:30,570 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,570 analyzer.py:main@148 [INFO] entrypoint 74f8b250
2020-09-07 19:55:30,571 analyzer.py:main@149 [INFO] maxexecs_addr 74f8b2ef
2020-09-07 19:55:30,571 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,572 analyzer.py:main@148 [INFO] entrypoint 74f902e0
2020-09-07 19:55:30,572 analyzer.py:main@149 [INFO] maxexecs_addr 74f90495
2020-09-07 19:55:30,572 analyzer.py:main@153 [INFO] <----------Scheduling------------>
```

```
2020-09-07 19:55:30,572 analyzer.py:main@148 [INFO] entrypoint 74f91590
2020-09-07 19:55:30,572 analyzer.py:main@149 [INFO] maxexecs_addr 74f915a5
2020-09-07 19:55:30,572 analyzer.py:main@148 [INFO] entrypoint 773e5b80
2020-09-07 19:55:30,573 analyzer.py:main@149 [INFO] maxexecs_addr 773e5b94
2020-09-07 19:55:30,573 analyzer.py:main@148 [INFO] entrypoint 773e5bd0
2020-09-07 19:55:30,574 analyzer.py:main@149 [INFO] maxexecs_addr 773e5be0
2020-09-07 19:55:30,574 analyzer.py:main@148 [INFO] entrypoint 77402c6a
2020-09-07 19:55:30,574 analyzer.py:main@149 [INFO] maxexecs_addr 77402c6a
2020-09-07 19:55:30,574 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,574 analyzer.py:main@148 [INFO] entrypoint 77402dd6
2020-09-07 19:55:30,574 analyzer.py:main@149 [INFO] maxexecs_addr 77402dd6
2020-09-07 19:55:30,574 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,574 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 19:55:30,575 analyzer.py:main@148 [INFO] entrypoint 77405aec
2020-09-07 19:55:30,575 analyzer.py:main@149 [INFO] maxexecs_addr 773de803
2020-09-07 19:55:30,575 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,575 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 19:55:30,576 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 19:55:30,576 analyzer.py:main@183 [INFO] <----------Shifting------------>
2020-09-07 19:55:30,577 analyzer.py:main@148 [INFO] entrypoint 7740617c
2020-09-07 19:55:30,577 analyzer.py:main@149 [INFO] maxexecs_addr 773cfb9c
2020-09-07 19:55:30,577 analyzer.py:main@153 [INFO] <----------Scheduling------------>
2020-09-07 19:55:30,577 analyzer.py:main@159 [INFO] <----------KRoundInit------------>
2020-09-07 19:55:30,577 analyzer.py:main@165 [INFO] <----------KRound------------>
2020-09-07 19:55:30,577 analyzer.py:main@183 [INFO] <----------Shifting------------>
2020-09-07 19:55:30,577 analyzer.py:main@190 [INFO] Collecting state bases
2020-09-07 19:55:30,590 analyzer.py:main@211 [INFO] Report overall stats
2020-09-07 19:55:30,590 analyzer.py:main@212 [INFO] entrypoint of comprehended caller 774065ea
2020-09-07 19:55:30,604 analyzer.py:main@223 [INFO] Collect & report routines/functions, max 3
     if there are duplicates
2020-09-07 19:55:30,645 analyzer.py:main@264 [INFO] Overall routine reporting
2020-09-07 19:55:30,701 analyzer.py:main@269 [INFO] Round routine is: 74f87a40 ..)
2020-09-07 19:55:30,704 analyzer.py:main@274 [INFO] Main function is: 74f87c60 ..)
2020-09-07 19:55:30,704 analyzer.py:main@280 [INFO] Collecting tap points data: states (
    ciphertexts or plaintexts)
2020-09-07 19:55:30,716 analyzer.py:main@295 [INFO] Collecting tap points data: key(s)
2020-09-07 19:55:30,718 analyzer.py:main@304 [INFO] Writing tap files (if any)
2020-09-07 19:55:30,720 analyzer.py:main@319 [INFO] Analyzer finished analyzing trace of
    638818822 bytes at 2020-09-07 19:55:30.720840
[omitted]
Running textprinter for TAP opensslaes256_774065ea_keys.tap and REPLAY opensslaes256 and ASID
    2ebe6000 with CipherTrace Analyzer's output
[omitted]
2020-09-07 21:43:42,409 verifier.py:main@63 [INFO] Data Series 1 was found in the read tap
    buffers, starting from line 716786 till 716792
2020-09-07 21:43:43,970 verifier.py:main@68 [INFO] Verifier finished searching 4 lines in
    buffers at 2020-09-07 21:43:43.970354
[omitted]
```

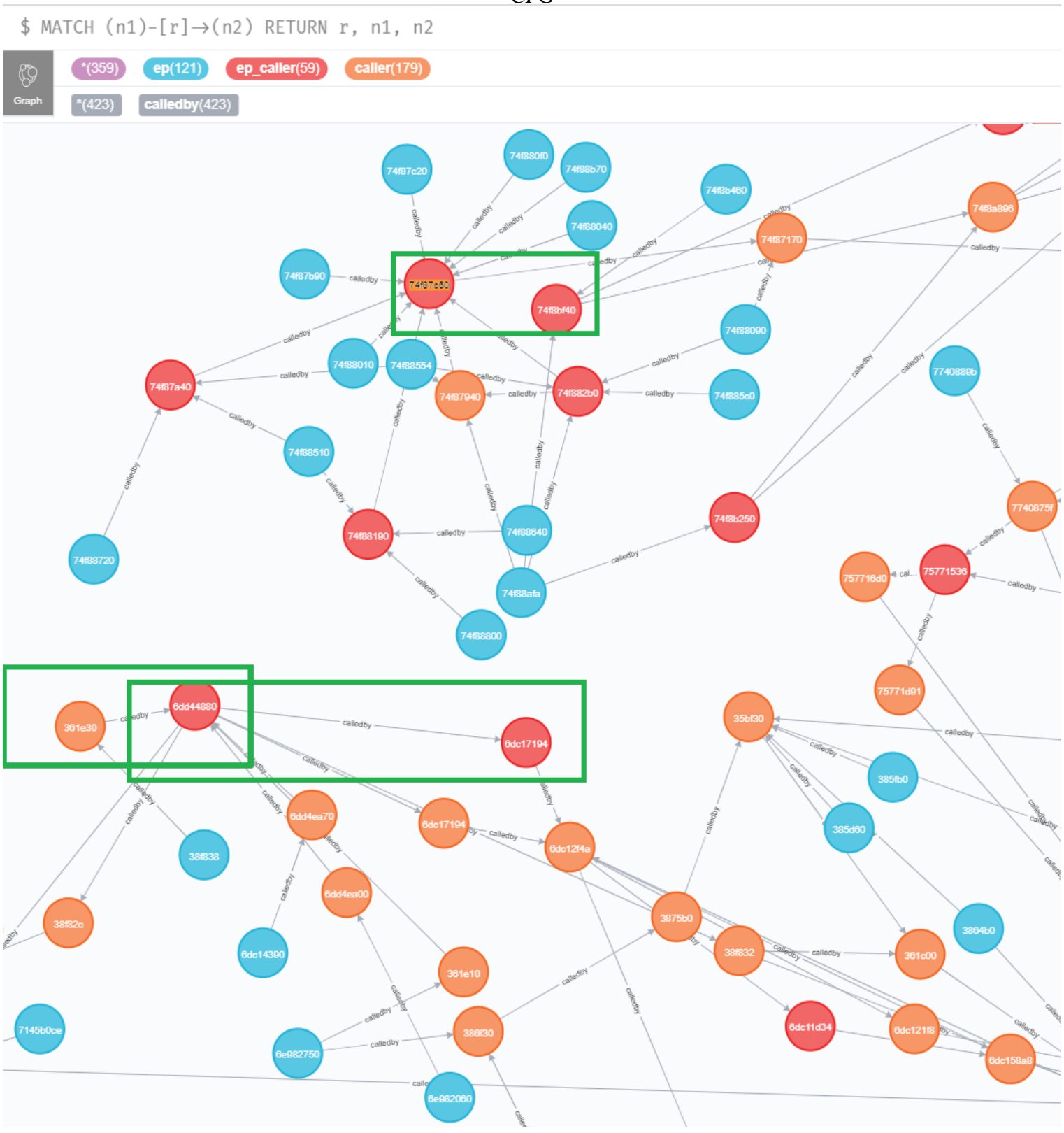

Figure A.6: OPENSSLAES256 CFG

*The rectangled ones are the main findings' main function and round routine.
*The highlighted one is the main finding's main function.

CALCPAINT_NOINTERNET

```
[omitted]
Running CipherTrace's Randometer with unigrams output
2020-09-06 10:09:58,365 randometer.py:main@45 [INFO] Starting Randometer for information
    measurement at 2020-09-06 10:09:58.365265
2020-09-06 10:09:58,365 randometer.py:main@46 [INFO] Reading unigram read file
    unigram_mem_read_report.bin
2020-09-06 10:10:05,462 randometer.py:main@52 [INFO] Reading unigram write file
    unigram_mem_write_report.bin
2020-09-06 10:10:11,168 randometer.py:main@60 [INFO] Computing randomness of read buffers
    using Chi-Squared test...
2020-09-06 10:10:11,551 randometer.py:main@63 [INFO] Computing randomness of write buffers
    using Chi-Squared test...
2020-09-06 10:10:11,894 randometer.py:main@68 [INFO] Computing read buffer entropy...
2020-09-06 10:10:13,179 randometer.py:main@71 [INFO] Computing write buffer entropy...
2020-09-06 10:10:14,219 randometer.py:main@74 [INFO] Entropy reads: 221728 writes: 178238
2020-09-06 10:10:14,219 randometer.py:main@77 [INFO] Applying read entropy mask > 0:
2020-09-06 10:10:14,349 randometer.py:main@81 [INFO] Applying write entropy mask > 0:
2020-09-06 10:10:14,421 randometer.py:main@87 [INFO] Applying read rand mask > 10000:
2020-09-06 10:10:14,502 randometer.py:main@91 [INFO] Applying write rand mask < 1000:
2020-09-06 10:10:15,357 randometer.py:main@105 [INFO] Results: reads: 2408, writes: 1223
[omitted]
2020-09-06 10:10:16,169 randometer.py:main@140 [INFO] Callers for ASID 5b679000 are set(['5
    f2a99', '5f2ad3', '743141a1'])
2020-09-06 10:10:16,172 randometer.py:main@142 [INFO] Randometer finished measuring
    information of unigram_mem_read_report.bin and unigram_mem_write_report.bin at 2020-09-06
    10:10:16.172718
[omitted]
Running CipherTrace's Analyzer for CALLER 5f2a99 and REPLAY calcpaint_nointernet and ASID 5
    b679000 with func_stats output
2020-09-06 10:42:02,100 analyzer.py:main@15 [INFO] Starting Analyzer for a 13386280901 bytes
    trace func_stats at 2020-09-06 10:42:02.100479
2020-09-06 10:42:02,100 analyzer.py:main@21 [INFO] filtering criteria from config: maxexecs,
    3, 1.0
2020-09-06 10:42:02,100 analyzer.py:main@34 [INFO] Reading the symbols file func_db
2020-09-06 10:42:02,135 analyzer.py:main@40 [INFO] Reading the stats file func_stats
Duration (excl. verifier): 215 minute(s) for REPLAY calcpaint_nointernet and ASID 5b679000 and
     CALLER 5f2a99
There are no tap files to check for caller 5f2a99 and replay calcpaint_nointernet
Duration: 215 minute(s) for REPLAY calcpaint_nointernet and ASID 5b679000 and Caller 5f2a99
Running CipherTrace's Analyzer for CALLER 5f2ad3 and REPLAY calcpaint_nointernet and ASID 5
    b679000 with func_stats output
2020-09-06 11:41:28,990 analyzer.py:main@15 [INFO] Starting Analyzer for a 13386280901 bytes
    trace func_stats at 2020-09-06 11:41:28.990238
2020-09-06 11:41:28,990 analyzer.py:main@21 [INFO] filtering criteria from config: maxexecs,
    3, 1.0
2020-09-06 11:41:28,991 analyzer.py:main@34 [INFO] Reading the symbols file func_db
2020-09-06 11:41:29,025 analyzer.py:main@40 [INFO] Reading the stats file func_stats
Duration (excl. verifier): 242 minute(s) for REPLAY calcpaint_nointernet and ASID 5b679000 and
     CALLER 5f2ad3
There are no tap files to check for caller 5f2ad3 and replay calcpaint_nointernet
Duration: 242 minute(s) for REPLAY calcpaint_nointernet and ASID 5b679000 and Caller 5f2ad3
Running CipherTrace's Analyzer for CALLER 743141a1 and REPLAY calcpaint_nointernet and ASID 5
    b679000 with func_stats output
2020-09-06 12:07:31,982 analyzer.py:main@15 [INFO] Starting Analyzer for a 13386280901 bytes
    trace func_stats at 2020-09-06 12:07:31.982068
2020-09-06 12:07:31,982 analyzer.py:main@21 [INFO] filtering criteria from config: maxexecs,
    3, 1.0
2020-09-06 12:07:31,982 analyzer.py:main@34 [INFO] Reading the symbols file func_db
2020-09-06 12:07:32,017 analyzer.py:main@40 [INFO] Reading the stats file func_stats
Duration (excl. verifier): 264 minute(s) for REPLAY calcpaint_nointernet and ASID 5b679000 and
     CALLER 743141a1
There are no tap files to check for caller 743141a1 and replay calcpaint_nointernet
Duration: 264 minute(s) for REPLAY calcpaint_nointernet and ASID 5b679000 and Caller 743141a1
[omitted]
Running CipherTrace's Randometer with unigrams output
2020-09-06 14:41:08,273 randometer.py:main@45 [INFO] Starting Randometer for information
    measurement at 2020-09-06 14:41:08.272953
2020-09-06 14:41:08,273 randometer.py:main@46 [INFO] Reading unigram read file
    unigram_mem_read_report.bin
2020-09-06 14:41:14,766 randometer.py:main@52 [INFO] Reading unigram write file
    unigram_mem_write_report.bin
2020-09-06 14:41:21,519 randometer.py:main@60 [INFO] Computing randomness of read buffers
    using Chi-Squared test...
2020-09-06 14:41:21,854 randometer.py:main@63 [INFO] Computing randomness of write buffers
    using Chi-Squared test...
2020-09-06 14:41:22,130 randometer.py:main@68 [INFO] Computing read buffer entropy...
2020-09-06 14:41:23,281 randometer.py:main@71 [INFO] Computing write buffer entropy...
2020-09-06 14:41:24,208 randometer.py:main@74 [INFO] Entropy reads: 221728 writes: 178238
2020-09-06 14:41:24,209 randometer.py:main@77 [INFO] Applying read entropy mask > 0:
```

```
[omitted]
2020-09-06 14:41:25,597 randometer.py:main@140 [INFO] Callers for ASID 5c4ab000 are set([])
2020-09-06 14:41:25,598 randometer.py:main@142 [INFO] Randometer finished measuring
    information of unigram_mem_read_report.bin and unigram_mem_write_report.bin at 2020-09-06
    14:41:25.598820
.....
Duration: 427 minute(s) for REPLAY calcpaint_nointernet and ASID 5c4ab000
```