# CAERUS: Chronoscopic Assessment Engine for Recovering Undocumented Specifications

Adam J. Seitz[1] Adam Satar[1] Brian Burke[1] Lok Yan[2]
Zachary J. Estrada[1]
*{adamjseitz@gmail.com,adamsatar@me.com,lok.yan@us.af.mil,estrada@rose-hulman.edu}*
[1]Rose-Hulman Institute of Technology, Terre Haute, IN USA
[2]Air Force Research Laboratory, Rome, NY USA

## Abstract

A significant feature of embedded systems, in particular legacy systems, is their sensitivity to signal timing. Any modifications (e.g., security protections) to legacy systems could affect the timing of critical control signals. Some timing properties are well known (e.g., baud rates for communication). However, other timing properties are not well specified or understood. Those properties only surface as a result of additional testing such as part of modernization or upgrade efforts. We present a programmable hardware/software framework to recover and uncover the undocumented timing properties of embedded systems, CAERUS. CAERUS is based on commodity hardware components and the software has been open sourced.

## 1 Introduction

Legacy systems are systems that are no longer supported by upstream vendors but must remain in use due to their criticality. Due to the lack of support, those legacy systems pose significant risk to security and maintenance efforts.[1] Many legacy systems provide critical functions, so removing them is often not an option. Modernizing legacy systems is challenging since much of the expertise and documentation related to those systems have been lost. Therefore, recovering undocumented properties of legacy systems is an important step towards modernization.

When trying to modernize a legacy system (e.g., by adding security features), it is expected that instead of replacing the system, new protection features may be introduced that treat the legacy code as a black box [5, 7]. Black box techniques are used in cases where the legacy system cannot be modified safely. For example, a legacy firmware binary might be executed in an emulator to preserve the behavioral semantics. However, embedded systems may also depend on precise timing for correct operation [1,3,4,6]. Additionally, the timing of those signals may not be well understood; the emulator will not be timing accurate leading to semantically equivalent but timing different behavior. Any external protection added to a legacy system may impact timing and therefore may impact the correct operation of that system. This dependence on timing becomes an undocumented specification, a specification that needs to be recovered. To summarize, we observe that while the logical behavior of legacy systems might be known (e.g., from test cases, characterizing good or correct behaviors from existing systems, or directly analyzing firmware binaries) how timing affects the behavior is unknown and must be recovered. In fact, we assume that the logical behavior is well known.

In this work, we present a new hardware analysis framework that is able to uncover timing properties of hardware circuits and/or communications buses, Chronoscopic Assessment Engine for Recovering Undocumented Specifications (CAERUS). CAERUS is based on commodity components and its code is open source.[2] The contributions of the CAERUS framework are the ability to: 1. record arbitrary digital signals, 2. programmatically interpret and perturb the signals both in terms of logic (e.g., voltage levels) and timing (e.g., jitter), and 3. replay the newly synthesized signals.

## 2 Related Work

Previous work on building tools for timing testbeds in hardware security has focused specifically on clock signals [2, 8]. There exist high-quality commercial (e.g., riscure Spider) and open source (e.g., NewAE ChipWhisperer) tools for performing timing side channel analysis in embedded systems. To the best of our knowledge CAERUS is the first open hardware record/replay platform targeted at uncovering the hidden timing properties of embedded systems.

[1] https://www.meritalk.com/articles/legacy-it-systems-obstacle-to-cybersecurity-gao-gene-dodaro/

[2] https://github.com/caerus-timing

## 3 Implementation

CAERUS has three main functionalities: signal capture, signal replay, and programmable signal perturbation. During signal capture, the system records input and output signals used by the device under test (DUT). During signal replay, the input signals stored during capture will be replayed to the device inputs. With the programmable perturbation functionality, signals can be modified both in terms of logic level and timing properties.

The physical design of CAERUS is illustrated in Figure 1. CAERUS has three main physical components: the signal playback system, the logic analyzer, and the command and control subsystem. The signal playback system is a custom hardware/software co-design which stores and replays signals with timing perturbations. The logic analyzer is a commodity device that is used by the command and control subsystem to record signals for later playback or for verification of functionality during testing.

### 3.1 Recording Signals

Signals are recorded using a Logic Analyzer, referred to as "LA" for the remainder of this paper. The Saleae Logic 8 LA used in this implementation provides a USB interface that can be controlled with a Python API.[3] This Python API is what the Command and Control (CnC) subsystem uses to control the LA. The CnC communicates with the LA to capture signals.

### 3.2 Replaying Signals

The signal playback subsystem (SPS) is implemented on a Digilent Zybo Zynq-Z7000 ARM/FPGA SoC development board. The Zybo board has an ARM core integrated with an FPGA. The SPS is capable of generating digital signal pulses on the order of nanoseconds and simultaneously driving up to four playback channels at variable frequencies. The SPS is composed of custom IP modules developed in Verilog HDL with accompanying firmware that exposes the replay functionality to the command and control subsystem. The CnC is responsible for mutating signals for playback. After modifying the previously recorded signals, the CnC delivers signals to the SPS. The signal representation in Python allows the user to define custom mutations. The list of mutations supported at the time of writing is shown in Table 1. Note that although the list of currently supported mutations is small, the Append mutation allows the user to construct an arbitrary signal out of smaller signals.

In addition to replaying signals, there are some advanced features that take into account the unique properties of embedded systems. The user can create arbitrary signals in the CnC user interface. CAERUS supports a "stop channel and stop address" feature. Stop channel and stop address allows

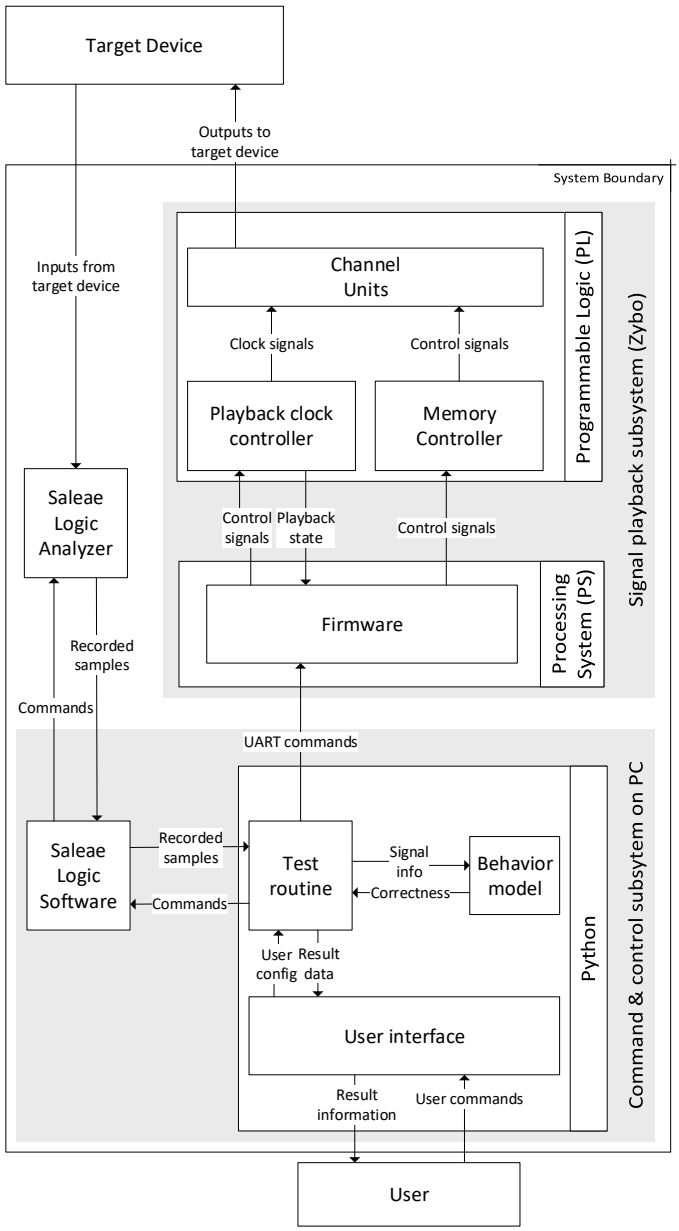
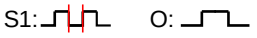
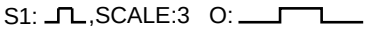
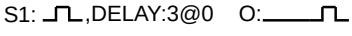
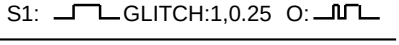



Figure 1: The design of CAERUS. The grey boxes outline the components developed as part of this work.

[3] https://pypi.org/project/saleae/

Table 1: Mutations Supported by CAERUS

| Operation | Description | Input(s) | Example |
|---|---|---|---|
| Append | Append another signal to this one | The other signal |   |
| Cut | Remove part of a signal | Start time/end time to remove |   |
| Scale X | Scale the rate of playback of the signal | Multiplier (scale factor) |   |
| Insert Delay | Insert a delay into the signal | Delay duration, delay start time |   |
| Glitch | Insert a glitch into the signal | Glitch start, glitch length |   |

the software to define an address where playback will stop as well as whether a particular signal should be played in a loop. This allows efficient playback of signals that are sampled at different rates (e.g., a clock on the order of MHz and a communication bus on the order of kbps).

## 3.3 Behavior Model

The goal of the experimental platform is to determine the sensitivity of an embedded system to variations in timing parameters of its input signals. The CnC system uses a behavior model to assert correctness of the system being tested. The behavior model describes the expected behavior of a system in terms of its signals. After an experiment, the CnC subsystem processes the recorded output signals according to the Behavior Model, if any of the conditions proscribed by the behavior model are violated, then that experiment fails verification. A description of how behavior models are defined is described in Section 4.4.

## 3.4 Built-in Tests

The expected use case of CAERUS is for the user to specify a test routine that is run by the CnC system. The test routine is responsible for coordinating and perturbing the replay of signals as well as running the behavior model to determine the timing properties of the system.

Along with supporting user-defined tests, CAERUS provides built-in test routines that could be used with a broad set of devices. These test routines still require the user to specify a behavior model to assert correct device behavior as correct behavior is target device dependent.

### 3.4.1 Clock Frequency Range

The clock frequency range test determines a precise range of valid clock frequencies for a device. The user inputs an upper and lower bound for the clock frequency. The test routine then uses a uses binary search with configurable precision to determine which clock frequencies still allow the behavior model to validate.

### 3.4.2 Duty Cycle Test

The duty cycle test allows the user to input a range of duty cycles along with a clock signal and a reset signal. This test reports a range of valid duty cycles for the clock, performing a binary search similar to the clock frequency range test.

### 3.4.3 Maximum Drift Test

This test finds the maximum delay that can be added between the beginning of two signals. Starting the signals offset is intended to simulate the effect of a drift that may have occurred over a longer period of time. The user specifies a maximum positive delay and a maximum negative delay as well as two input signals.

### 3.4.4 Maximum Glitch Duration

The maximum glitch duration test allows the user to test the resilience of a system to glitches in the clock signal. The inputs of the test are: a clock signal, a reset signal, and a location value in the range [0,100] to specify where the glitch will begin in the clock cycle.

# 4 Usage

The general process for running a test through CAERUS is: define a test routine, define a behavior model, record inputs, and run the test routine. The user interacts with CAERUS through a software interface that communicates with the CnC system. The CnC system then communicates with both the logic analyzer and the record/replay device to perform the steps outlined previously.

## 4.1 Assumptions

In order to use CAERUS, the user must understand the correct functional behavior of the test system in terms of its measurable inputs and outputs. This functional behavior is captured by the user in the behavior model (e.g., if the reported vehicle speed is $\leq$ 30mph, a diesel engine controller should disable engine braking). For many experiments, a range/space of possible timing values is also provided by the user. E.g., a user inputs that the maximum clock frequency for a MCU will be

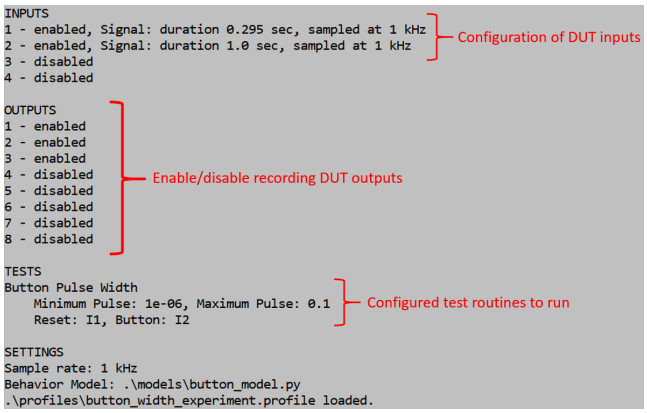



Figure 2: An example of the user interface when running an experiment. DUT is Device Under Test

within the range [20,40Mz] when performing a clock range test from Section 3.4.1.

The main goal of CAERUS is to uncover the timing permutations that cause the system to stop behaving correctly. It may be worth noting that in many cases, the actual response of a system to incorrectly timed input(s) may also be unknown to the user. That is, the behavior model will fail, but how exactly the outputs would change is not known a priori.

## 4.2 The User Interface

A screenshot of CAERUS's interface is shown in Figure 2. This screenshot was taken during the setup of the experiment presented in Section 5. In Figure 2, we see multiple fields. The INPUTS/OUTPUTS fields describe the inputs and outputs from the device under test (DUT) perspective. The outputs correspond to the possible channels that could be recorded from the Logic Analyzer (8 channels are supported as we are using the Saleae Logic 8). The TESTS list the test routine(s) currently selected to run. The SETTINGS describe the configuration of the LA, such as the sample rate, as well as which behavior model is used for asserting correct device behavior.

## 4.3 Defining a Test Routine

The test routine is a python script responsible for coordinating the replay of signals, applying perturbations, and running the behavior model. A test routine will typically run for multiple iterations, sweeping various parameters. As seen in the sample test routines given in Section 3.4, the mutations applied to the signal change dynamically during a test (e.g., to find some upper or lower bound for a given timing property).

## 4.4 Defining a Behavior Model

The behavior model's inputs are the input and output signals from the target device. The behavior model checks those signals to assert if the system performed its task successfully, reporting the result back to the test routine. The test routine then modifies the signals and executes the next iteration based on the result from the behavior model.

The behavior model does not need to capture the complete functionality of the test system, only what is needed to verify correct operation in a given test. The behavior model corresponds to the expected behavior of the target device from a black box perspective. Having access to the binary/source code of a test system significantly improves development of the behavior model. If a reference contains timing information (e.g., maximum clock frequency), that documentation could be used to determine ranges. CAERUS would then be used to determine the physical limits (e.g., how does the system actually behave).

We note that defining a behavior model for a complex system is nontrivial. The behavior model can be represented abstractly by a state machine, with the pattern of signals providing the state transitions. For very complex systems, we expect the behavior model would be developed iteratively with the test routine, helping the user understand their device. In this way, we imagine that CAERUS could also be helpful as a reverse-engineering tool that allows a user to build a software model of a hardware device. Automatically generating a behavior model is beyond the scope of this work.

# 5 Example

## 5.1 Minimum Button Duration

We present a simple example to illustrate the usage of CAERUS in uncovering a timing property of a system. The goal of this example experiment is to determine the minimum time needed for a system to register a button press.

### 5.1.1 Device Operation

We used a PIC16F887 microcontroller with one digital input (a button) and one UART output. We wrote a simple software program to respond to the digital input as a button press. When the button is pressed, the output signal is raised high by the MCU only if the button has been held down for a certain amount of time (software debounce). The amount of time the button needs to be held for the button press to register is software configurable and we change this parameter in our experiments.

### 5.1.2 Test Routine and Behavior Model

The goal of the experiment is to uncover the minimum duration that will still cause the MCU to register the button press. The behavior model checks to see if output signal is logic high after the button is pressed. If the signal is high, then the system registered the button press. The test routine does a

Table 2: Minimum Button Duration

| Duration (ms) | Mean | StdDev | Min/Max |
|---|---|---|---|
| 1 | 1.005 | $2.985\times10^{-3}$ | 1.001/1.007 |
| 7 | 7.000 | $6.569\times10^{-3}$ | 6.993/7.055 |
| 34 | 34.00 | $8.413\times10^{-3}$ | 33.97/34.01 |
| 1 - HS | 1.026 | 0 | 1.026/1.026 |
| 7 - HS | 7.024 | 0 | 7.024/7.024 |
| 34 - HS | 34.04 | $1.194\times10^{-4}$ | 34.02/34.88 |

binary search over pulse widths until it finds the minimum pulse with a certain error (10$\mu$s in our experiments). Once the behavior model fails to validate, the test routine backs off until it finds a pulse width that does validate. That pulse width is reported as the minimum button duration.

#### 5.1.3 Results

The button durations tested were 1ms, 7ms, and 34ms. Those values were chosen as 7ms is a typical debounce time and 1ms and 34ms represent extreme values. We ran 100 experiments for each button press duration and the results are in the Table 2. Two oscillators were used: the internal RC oscillator (accuracy of $\pm$ 1%) and an external HS crystal oscillator (tolerance of $\pm$30ppm). In Table 2, we see that the system was able to correctly identify the duration for each configuration.

We see that the standard deviation is larger for the longer durations with the internal oscillator. These variations are expected due to accumulated error as longer durations require more timer ticks. This is evidenced by the much lower standard deviation with the higher precision oscillator. The HS results did have a mean that was further from the ideal value. The difference from ideal might be explained by the overhead in processing interrupts or lack of clock synchronization between the SPS and DUT. This simple test demonstrates that timing behaviors are rarely straightforward and we plan to explore further in future work. CAERUS allows us to perform that exploration programmatically and economically.

## 6 Conclusions

We have presented CAERUS, an open platform for uncovering timing properties of embedded systems. The system uses a custom hardware/software platform built on commodity components. CAERUS performs record/replay of signals, supporting a variety of mutations and built-in tests. The platform was designed with extensibility in mind and provides a Python API for users to create their own tests and validation mechanisms. In future work, we will use CAERUS to uncover the timing properties of complex systems as well as validate the security impact of timing violations. We will also develop more advanced analysis capability and improve the user experience.


## Acknowledgments

The authors wish to thank the anonymous reviewers as well as Stephen Schwab for their insightful feedback.

Effort sponsored by the Air Force under MOU FA8750-15-3-6000. The U.S. Government is authorized to reproduce and distribute copies for Governmental purposes notwithstanding any copyright or other restrictive legends. Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the US Air Force and/or US government. Approved for public release: #88ABW-2019-2592.


## References


[1] Matthias Becker, Saad Mubeen, Moris Behnam, and Thomas Nolte. Extending automotive legacy systems with existing end-to-end timing constraints. In Shahram Latifi, editor, *Information Technology - New Generations*, pages 597–605, Cham, 2018. Springer International Publishing.

[2] Santosh Desiraju. *High Speed Clock Glitching*. PhD thesis, Cleveland State University, 2015.

[3] Peter A Sandborn and Varun J Prabhakar. The forecasting and impact of the loss of critical human skills necessary for supporting legacy systems. *IEEE Transactions on Engineering Management*, 62(3):361–371, 2015.

[4] Alberto Sangiovanni-Vincentelli and Marco Di Natale. Embedded system design for automotive applications. *Computer*, 40(10):42–51, 2007.

[5] Dries Schellekens, Brecht Wyseur, and Bart Preneel. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming*, 74(1-2):13–22, 2008.

[6] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[7] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168. ACM, 2012.

[8] Christian Wenzel-Benner and Jens Gräf. Xbx: external benchmarking extension for the supercop crypto benchmarking framework. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 294–305. Springer, 2010.